Biblioteczka **Komputer Świat**

KSIĄŻKA Z PŁYTĄ DVD



ZACZNIJ PROGRAMOWAĆ

nauka kodowania w prostych projektach

C#

PORADY KROK PO KROKU: JAK TWORZYĆ GRY W C#

Z TEJ KSIĄŻKI NAUCZYSZ SIĘ, JAK:

■ posługiwać się językiem C# ■ tworzyć proste gry w C# ■ wykorzystywać programowanie obiektowe ■ tworzyć gry 3D w Unity ■ programować na urządzeniu mobilnym

NA DVD ZNAJDZIESZ NAJLEPSZE: ■ edytory kodu ■ środowiska programistyczne ■ silniki do tworzenia gier



Z TĄ KSIĄŻKĄ E-WYDANIE GRATIS

Poniżej znajduje się płyta z kodem bonusowym dającym dostęp do e-wydania tej książki w serwisie KŚ+ (www.ksplus.pl) pliku ISO z obrazem załączonej płyty do pobrania.

NIEZBĘDNIK PROGRAMISTY

Na płycie znajdziemy najlepsze edytory kodu źródłowego, środowiska programistyczne do tworzenia programów, gier i aplikacji mobilnych, a także silniki do tworzenia gier 2D i 3D na PC, smartfon i tablet.

Jeśli brakuje płyty, poinformuj sprzedawcę lub redakcję: redakcja@komputerswiat.pl



Kod bonusowy należy zarejestrować w KŚ+ (www.ksplus.pl)

Biblioteczka Komputer Świat

KONRAD JAGACIAK

ZACZNIJ PROGRAMOWAĆ

nauka kodowania w prostych projektach

C#

ringier
axel springer



AUTOR: Konrad Jagaciak

REDAKTORZY PROWADZĄCY: Rafał Kamiński, Agnieszka Al-Jawahiri, Krzysztof Dziedzic

PRZYGOTOWANIE PŁYTY: Mariusz Michalski

PROJEKT OKŁADKI: Robert Dobrzyński

SKŁAD I ŁAMANIE: Robert Dobrzyński, Mariusz Rybak

KOREKTA: Jolanta Rososińska

WYDAWCA: RINGIER AXEL SPRINGER POLSKA Sp. z o.o.
02-672 Warszawa, ul. Domaniewska 52
tel. 22 2320000, 22 2320001
www.ringieraxelspringer.pl

ISBN: 978-83-8091-626-5

Warszawa 2018

DYREKTOR WYDAWNICZY: Paweł Paczuski

BUSINESS PROJECT MANAGER: Paweł Bulwan

DRUK I OPRAWA: Drukarnia im. Adama Półtawskiego, Kielce

EGZEMPLARZE ARCHIWALNE:
www.literia.pl

tel. 22 3367901
infolinia 801 000 869

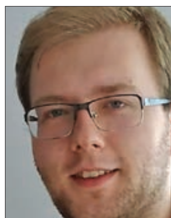
E-WYDANIA: www.ksplus.pl

KONTAKT:
redakcja@komputerswiat.pl

INTERNET: komputerswiat.pl, ksplus.pl

ringier
axel springer





Konrad Jagaciak
programista
prowadzący zajęcia
komputerowe
z młodzieżą

od autora

Pisząc tę książkę, myślałem o wszystkich podręcznikach do nauki programowania, które miałem możliwość przeczytać. Zastanawiałem się, czego mi w nich brakowało. Postanowiłem stworzyć książkę taką, z jakiej sam chciałbym się uczyć.
Wiem z doświadczenia, że nic nie uczy tak jak praktyka. Dlatego przedstawiam instrukcje programistyczne, opisując tworzenie gier w C# (za pomocą Visual Studio Community 2017). Właśnie na przykładach konkretnych, prostych projektów nauczymy się wykorzystywać programowanie obiektowe, stosować instrukcje warunkowe i pętle, a także zobaczymy, w jaki sposób komunikować się w pisanych przez nas grach z ich użytkownikami – graczami. A na deser przeczytamy, jak ułożyć grę na Androida oraz jak „działa" C# w znanym darmowym narzędziu do tworzenia efektownych gier 2D i 3D – Unity.
Książka ta prezentuje podstawy programowania w języku C# – jednak zdobędziecie dzięki niej także uniwersalną wiedzę programistyczną, którą możecie wykorzystać w przyszłości, poznając kolejne języki programowania.

Zapraszam do lektury!

PROGRAMY OPISANE W TYM ROZDZIALE ZNAJDZIESZ NA DVD I W KŚ+

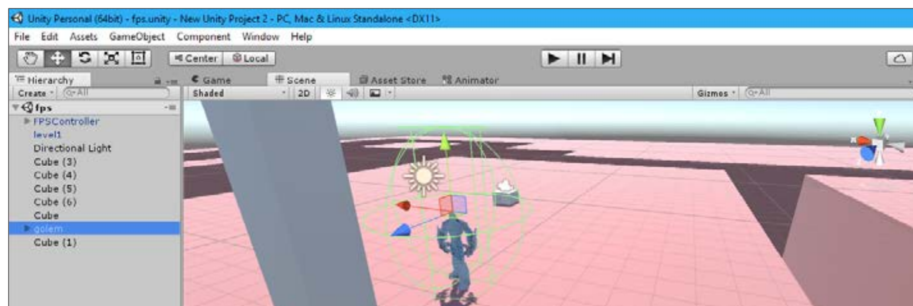
1 Na czym polega platforma .NET

Decydując się na rozpoczęcie przygody z programowaniem, powinniśmy mądrze wybrać technologię, którą poznamy jako pierwszą. Powinna być ona dobrym punktem wyjściowym do dalszego rozwoju

Od czego zacząć

W świecie programowania istnieje wiele technologii, narzędzi, języków, a nawet sposobów na rozumienie i postrzeganie programowania. Sposób patrzenia programisty na proces programowania nazywamy **paradygmatem**. Obecnie popularnym paradygmatem jest **paradygmat obiektowy** – czym się charakteryzuje i różni od innych, przeczytamy w kolejnym rozdziale.

Wybór paradygmatu obiektowego, jako podejścia, które będziemy poznawać, ma wpływ na to, spośród których środowisk będziemy wybierać. Wśród **obiektowych języków programowania** wymienić można: **Javę**, **C#**, **Visual Basic.NET** czy **C++**. Trzy ostatnie coś łączy. To **platforma .NET Framework**, w ramach której można tworzyć programy właśnie za pomocą tych języków.



Na język C# warto postawić choćby dlatego, że za jego pomocą możemy tworzyć gry 3D przy wykorzystaniu popularnych silników gier, jak Unity i Godot

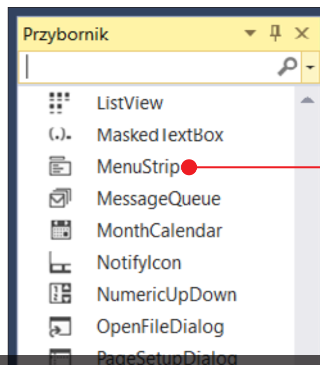
Platforma .NET

Można powiedzieć, że .NET jest zbiorem bibliotek, które zdecydowanie ułatwiają nam programowanie. Nie będzie to jednak pełna definicja platformy .NET. Bardzo ważną jej częścią jest **CLR** – czyli wspólne środowisko uruchomieniowe. Jest ono podstawą funkcjonowania platformy .NET. To właśnie dzięki niemu możliwe jest wykorzystywanie tego samego zbioru funkcji zawartych w bibliotekach przez programy napisane w różnych językach programowania.

Skupmy się jednak na tym, co dają nam biblioteki zawarte w platformie .NET. Znajdziemy w niej biblioteki pozwalające na tworzenie aplikacji internetowych (**Web Forms**), aplikacji dla systemu Windows (**Windows Forms**) czy usług internetowych (**Web Services**), ale także inne. W tej książce skupimy się na samodzielnych aplikacjach dla systemu Windows.

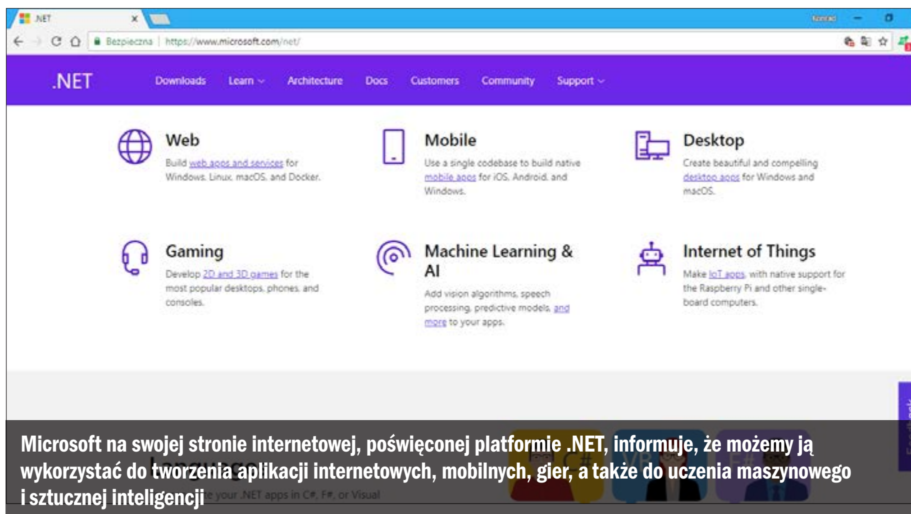
Windows Forms

Windows Forms to nazwa nawiązująca do interfejsu programowania graficznych aplikacji w ramach Microsoft .NET Framework. Daje on nam dostęp do elementów interfejsu graficznego MS Windows – czyli przycisków,



Elementy, które chcemy wykorzystać w oknie tworzonej przez nas aplikacji, wybieramy z Przybornika. Jeśli chcemy wykorzystać menu, wystarczy, że przeciągniemy kontrolkę MenuStrip do modelowanego przez nas okna aplikacji, a następnie określimy, jakie przyciski chcemy w nim umieścić

pól tekstowych czy innych obiektów często widywanych przez nas w oknach popularnych programów, jak choćby pasek menu u góry okna. Mimo że aplikacje Windows Forms są stopniowo wypierane przez inne typy – jak WPF – dobrze jest się z nimi zapoznać, gdyż stanowią doskonałą podstawę do późniejszego tworzenia innych typów aplikacji.



Microsoft na swojej stronie internetowej, poświęconej platformie .NET, informuje, że możemy ją wykorzystać do tworzenia aplikacji internetowych, mobilnych, gier, a także do uczenia maszynowego i sztucznej inteligencji

Język

Uczymy się programowania, aby móc w przyszłości tworzyć wielkie projekty i rozbudowane gry. Wybierając jeden spośród języków z platformy .NET, powinniśmy zastanowić się, który z tych języków najbardziej przyda nam się w przyszłości. Na pewno warto postawić na **język C#**. Jest on wykorzystywany między innymi w Unity, czyli największym darmowym silniku pozwalającym na tworzenie gier 3D.

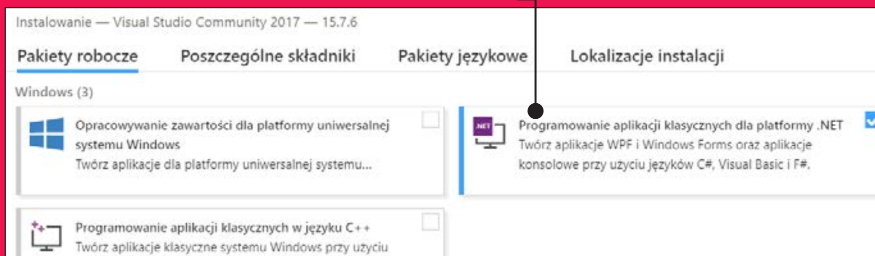
Narzędzie

Do tworzenia aplikacji Windows Forms wykorzystamy program **Visual Studio Community 2017** (znajdziemy go na płycie – **DVD-KOD: 025**).

3 Następnie możemy określić, że tworzyć będziemy aplikację **Windows Forms**. Podajemy też nazwę i lokalizację nowo utworzonego projektu i klikamy na **OK**.

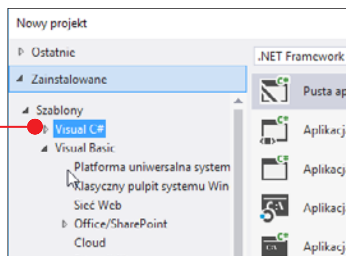
POTRZEBNE PAKIETY

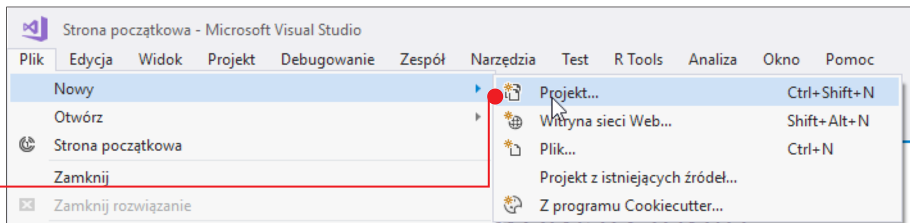
Podczas instalacji Microsoft Visual Studio trzeba pamiętać o pakietach, które będą potrzebne podczas używania programu. Wystarczy, że zaznaczymy opcję Programowanie aplikacji klasycznych dla platformy .NET i klikniemy na Zainstaluj.

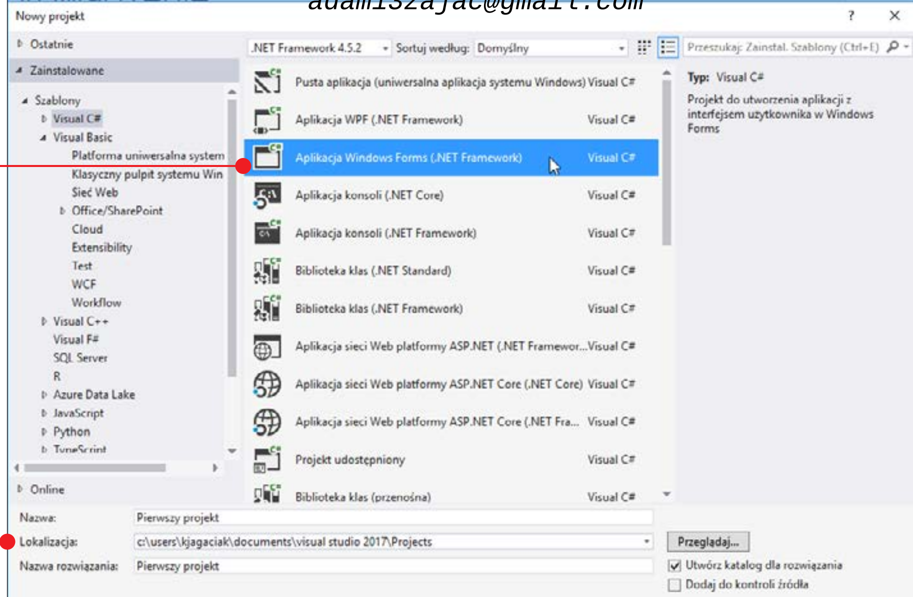


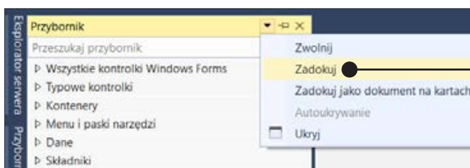
1 Po uruchomieniu programu z menu głównego wybieramy **Plik**, następnie **Nowy**, a na końcu **Projekt**.

2 Otwarte zostanie okno kreatora nowego projektu. Najważniejsze dla nas w tym momencie jest to, aby w szablonach, widocznych w drzewie po lewej stronie okna, wybrać język **Visual C#**.





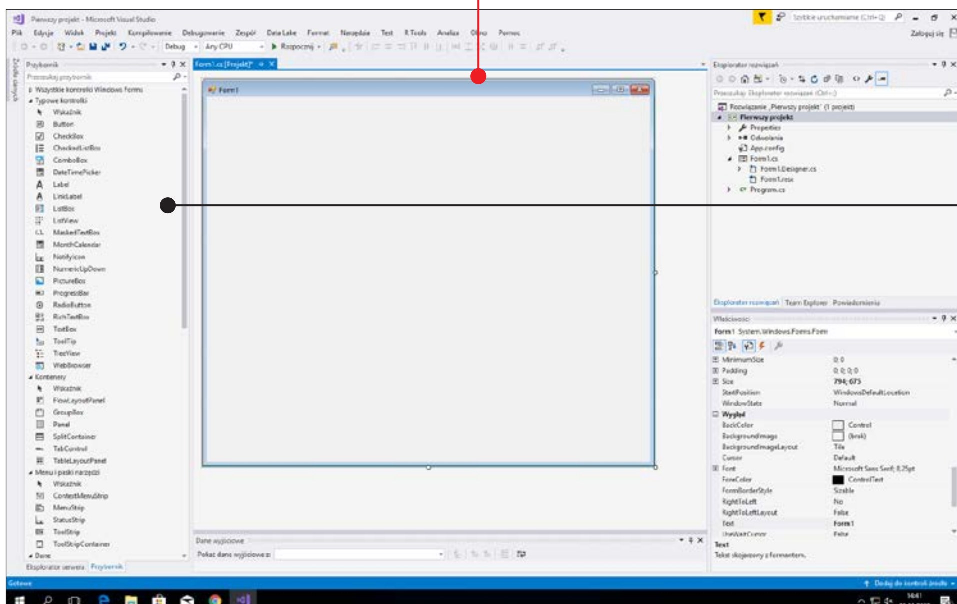




PRZYBORNIK

Na początku korzystania z programu warto dodać na stałe panel **Przybornika** ●. Robimy to, klikając po lewej stronie na **Przybornik**, następnie na ikonę strzałki w dół i wybierając opcję **Zadokuj** ●.

4 Otworzy się wtedy okno pustego projektu ●, w którym będziemy mogli rozpocząć pracę nad naszą pierwszą grą w języku C#. A żeby móc to zrobić, w kolejnym rozdziale poznamy podstawy języka, a także podstawy paradygmatu obiektowego.



PROGRAMY OPISANE W TYM ROZDZIALE ZNAJDZIESZ **NA DVD I W KŚ+**

2 Od Scratcha do C#

Scratch to doskonałe narzędzie do nauki programowania nie tylko dla dzieci, ale również dla dorosłych. Dzięki niemu łatwiej będzie nam zrozumieć paradygmat obiektowy

Każdy, kto interesuje się tworzeniem gier, na pewno miał okazję poznać najpopularniejsze narzędzie edukacyjne do nauki programowania, jakim jest Scratch. Pozwala on samodzielnie tworzyć własne gry w wizualnym i prostym środowisku. Scratch stanowi świetną podstawę do dalszego poznawania profesjonalnych języków programowania. Zawarte w nim bloki doskonale odpowiadają instrukcjom programistycznym, jakie musimy poznać, aby programować w języku C#.



KSIĄŻKI DO NAUKI PROGRAMOWANIA

Dwie książki z serii Biblioteczka Komputer Świata dla początkujących programistów można kupić w sklepach internetowych: Literia – pod adresem **literia.pl** (wydania papierowe) oraz KŚ+ – pod adresem **www.ksplus.pl** (e-wydania do czytania w oknie przeglądarki i do pobrania w formacie PDF).

Komunikacja z graczem

Podstawą w tworzeniu gier jest komunikacja z graczem. Powinniśmy wiedzieć, jak przekazywać graczowi informacje (w formie tekstu), a także jak pytać gracza o istotne dla przebiegu gry kwestie, na przykład o to, jak ma na imię, aby nasza gra wiedziała, jakim imieniem zwracać się do użytkownika. W Scratchu do wyświetlania tekstów

w oknie gry służą bloki **powiedz** i **pomyśl** ●. W języku C# podobne działanie możemy

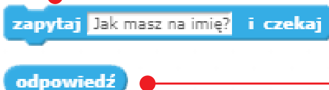


osiągnąć poleceniem **Console.WriteLine();**. Zarówno widoczny blok, jak i polecenie ● – wyświetlą napis „Dzień dobry".

```
Console.WriteLine("Dzień dobry");
```



Aby pobierać dane od gracza, w Scratchu mamy do dyspozycji blok **zapytaj** ●. Działa



on tak, że wyświetlane jest puste pole tekstowe i dymek z treścią pytania umieszczoną przez nas w bloku ●. To co gracz wpisze do pola tekstowego, trafia wprost do bloku **odpowiedź** – a my, programując, możemy korzystać z tego bloku jak z wartości zmiennej ●. W C# mamy do dyspozycji polecenie **Console.ReadLine();**. Nie mamy jednak zmiennej **odpowiedź**, do której standardowo trafiałoby to, co zostanie wpisane przez gracza. Zmienną, do której trafi wpisana przez gracza treść, musimy zadeklarować sami (określając przy tym jej typ). W omawianym poleceniu nie mamy też możliwości wyświetlenia tekstu, który podpowiadałby graczowi, co należy wpisać. Dlatego, aby poinformować gracza o konieczności podania jakiejś wartości (czyli w celu uzyskania efektu znanego ze Scratcha), powinniśmy najpierw użyć polecenia **Console.Writeline();**, a dopiero potem wczytywać wartość zmiennej ●.

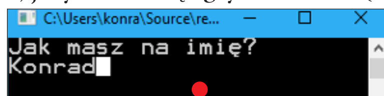



```
Console.WriteLine("Jak masz na imię?");
string str;
str = Console.ReadLine();
```

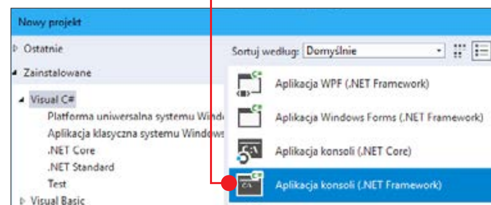
ZMIENNA STRING

Linijka kodu **String str;** składa się z typu zmiennej, czyli słowa **String** i jej nazwy, w tym wypadku **str**. O ile nazwy zmiennych mogą być dowolne, o tyle ich typy są ściśle określone. Do przechowywania danych tekstowych będziemy zawsze tworzyć zmienne w typie string.

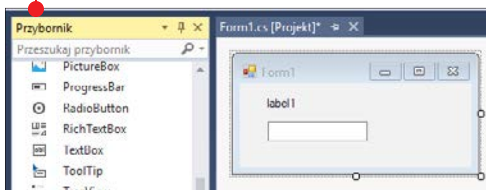
W dwóch poznanych właśnie poleceniach powtarza się słowo **Console**. Odnosi się ono do konsoli. Z tych poleceń możemy korzystać, jedynie tworząc gry konsolowe (czyli



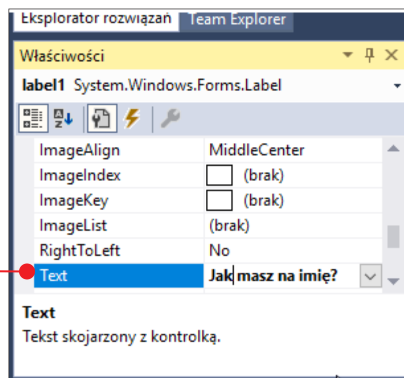
w oknie tekstowym ●). Aby utworzyć taki program, tworząc nowy projekt powinniśmy wybrać **Aplikacja konsoli (.NET Framework)** ●, zamiast Windows Forms. Nie



wykorzystamy tych poleceń do komunikacji z graczem w programie okienkowym.
Wygląd programu okienkowego budujemy z kontrolek. Kontrolki znajdują się w **Przyborniku** widocznym domyślnie po lewej stronie okna programu. Aby wpisać w oknie programu jakiś tekst, powinniśmy umieścić w programie odpowiednią kontrolkę, która na to pozwala. Podobnie będzie z polem, w którym gracz będzie mógł coś napisać. Do wpisywania najczęściej używane są etykiety (kontrolka **Label**), a do pobierania danych – pola tekstowe (kontrolka **Textbox**). Kontrolki umieszczamy w oknie programu, przeciągając je do okna z przybornika. Początkowo etykieta nie wygląda jednak tak, jak powinna.



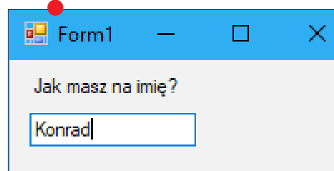
Aby znalazł się w niej potrzebny nam tekst, powinniśmy zmienić wartość właściwości **Text**. Możemy to zrobić poprzez okno **Właściwości**, wpisując tekst w odpowiedniej rubryce, lub też z poziomu kodu.
Wartości właściwościom przypisujemy tak: najpierw podajemy nazwę elementu okna programu, następnie nazwę właściwości, po czym po znaku równości, który pełni tu rolę przypisania, dopisujemy wartość.



W naszym przykładzie, gdy dodana przez nas etykieta nosi nazwę **Label1** – możemy zmienić jej tekst następującym poleceniem: **Label1.Text = „Jak masz na imię?”**.
Również pola tekstowe mają właściwość **Text**, której wartość może być edytowana przez gracza podczas użytkowania programu. Wtedy do interesującej nas zmiennej powinniśmy zapisać to, co aktualnie znajduje się pod właściwością **Text** w polu tekstowym.

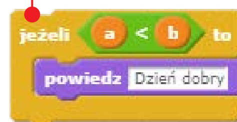
```
String str;
str = textBox1.Text;
```

A to efekt.



Instrukcja warunkowa

Niezbędnym elementem języków programowania jest instrukcja warunkowa. Jest to polecenie, które pozwala na wykonywanie określonych czynności w zależności od tego, czy zdefiniowany przez programistę warunek (wyrażenie logiczne) jest spełniony, czy nie. Z instrukcją warunkową możemy spotkać się także w Scratchu. Jej rolę pełni blok **jeżeli**. Bloki, które umieścimy we-

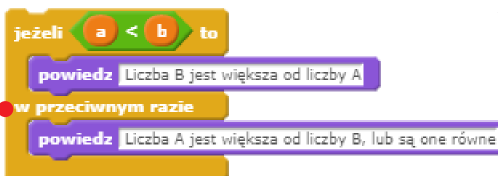


wnątrz niego, będą wykonane tylko wtedy, gdy prawdą jest to, co umieściliśmy w sześciokątnym polu wyrażenia. W języku C# instrukcja warunkowa to instrukcja **If**. W niej również podajemy warunek, jaki ma zostać spełniony, aby wykonane zostały polecenia umieszczone wewnątrz instrukcji – gdzie wnętrze instrukcji zawiera się w środku nawiasu klamrowego.

```
if (a > b)
{
    label1.Text = "Dzień dobry";
}
```

A co się stanie, jeśli warunek ten nie zostanie spełniony?

W Scratchu mamy blok, który pozwala wykonać jedne instrukcje w przypadku spełnienia warunku, a inne w przypadku, gdy nie jest on spełniony.



W C# również mamy polecenie pozwalające na wykonanie pewnych instrukcji, gdy podany przez nas warunek nie jest spełniony – do instrukcji **If** należy dopisać część **else** A, a następnie umieścić wewnątrz nawiasu klamrowego polecenia, które należy wykonać.

W C# nie jest to jednak jeszcze koniec instrukcji warunkowej. Jedna instrukcja może sprawdzać po kolei kilka warunków. Jeśli

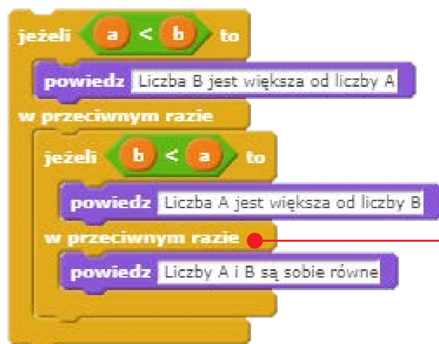
```
if(a<b)
{
    label1.Text = "Liczba B jest większa od liczby A";
}
else if (a>b)
{
    label1.Text = "Liczba A jest większa od liczby B";
}
else
{
    label1.Text = "Liczby są sobie równe";
}
```

pierwszy nie zostanie spełniony, sprawdzi drugi warunek. A jeżeli ten też nie będzie spełniony, sprawdzony zostanie trzeci i tak dalej.

Gdy żaden z warunków nie będzie spełniony, wykonane zostaną polecenia umieszczone w części **else**.

Aby dodać kolejne warunki do instrukcji, dopisujemy części **else if** B – części **else if** może być wiele w jednej instrukcji warunkowej.

Analogicznie do tego przykładu, aby zrealizować to samo działanie w Scratchu, musielibyśmy umieścić blok **jeżeli-w przeciwnym razie** w bloku **jeżeli-w przeciwnym razie**.



Pętle

Kolejne ważne instrukcje programistyczne to pętle. W Scratchu znajdziemy ich kilka – są to bloki oznaczone u dołu strzałką skierowaną ku górze.



Oznacza ona, że po wykonaniu wszystkich instrukcji umieszczonych w pętli przechodzimy do pierwszej instrukcji i wykonujemy instrukcje kolejny raz. Każda z pętli charakteryzuje się innym sposobem wyznaczania końca pętli.
Pętla **zawsze** nie ma warunku, zatem jej wykonywanie może zakończyć tylko koniec programu bądź wyłączenie skryptów.
Pętla **powtórz X razy** będzie wykonywana dokładnie tyle razy, ile podamy.
Pętla **powtarzaj aż** będzie wykonywała się, dopóki spełniony będzie warunek w niej umieszczony. Skupmy się na dwóch ostatnich z wymienionych pętli.
Również w C# znajdziemy pętlę, która może wykonać się określoną liczbę razy. Jest to pętla **for**. Do jej skonstruowania niezbędne jest stworzenie działającej w jej obrębie zmiennej, która z każdym przejściem pętli będzie zwiększała swoją wartość o podany przez nas krok. Koniec pętli wyznacza podany przez nas warunek.

```
for (int i=0;i<10;i++)
{

}
```

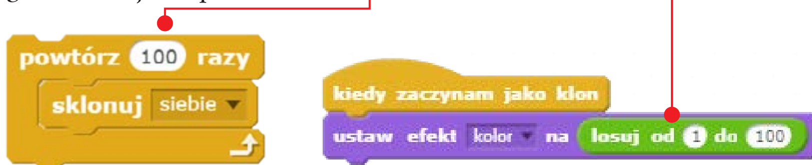
Nasza przykładowa pętla wykona się 10 razy. W jej pierwszym wykonaniu zmienna **i** będzie równa **0**, w kolejnym **1** (dlatego że instrukcja **i++** oznacza zwiększenie wartości zmiennej **i** o **1**), w kolejnym **2** i tak dalej. Pętla wykonuje się do momentu, gdy warunek przestanie być spełniany – ostatnia wartość **i**, kiedy warunek jest spełniony, to **9**. Jak łatwo policzyć, da nam to 10 wykonań pętli. Takie przejścia pętli nazywane są **iteracjami**.
Mamy też do dyspozycji pętlę, której wykonywanie zależne jest od osiągniętego efektu. Pętla **while** jest analogią do pętli **powtarzaj aż** w Scratchu. Pętla ta będzie wykonywała umieszczone w nawiasie klamrowym instrukcje, dopóki prawdą będzie warunek, jaki w niej zapiszemy. Instrukcje wewnątrz pętli powinny modyfikować wartości zmiennych, których użyliśmy do budowy warunku. Jeśli wartości tych zmiennych nie będą się zmieniać, a warunek pozostanie cały czas spełniony, pętla nie przestanie się wykonywać, działając niczym pętla **zawsze** ze Scratcha.

```
while (a > b)
{

}
```

Programowanie obiektowe

Programowanie obiektowe to pojęcie rozbudowane i skomplikowane. Ta książka pomoże je zrozumieć. A oto, co musimy wiedzieć na samym początku: **programowanie obiektowe wymaga od nas definiowania klas i inicjalizacji ich obiektów**. Wbrew pozorom nie jest to wcale takie trudne. W poznaniu tego zagadnienia znów pomoże nam Scratch.
Wyobraźmy sobie, że musimy stworzyć grę, w której gracza atakuje stu przeciwników. Każdy z nich ma takie samo zadanie do wykonania – biec w stronę gracza i uderzać go mieczem.
Każdy z przeciwników może mieć inny kolor miecza. Tworząc taką grę, nie zadawalibyśmy sobie przecież aż tyle trudu, aby projektować każdego przeciwnika z osobna.
W Scratchu moglibyśmy „napisać” jednego przeciwnika, a następnie tworzyć jego klony. A każdy klon dostawałby losowy kolor.

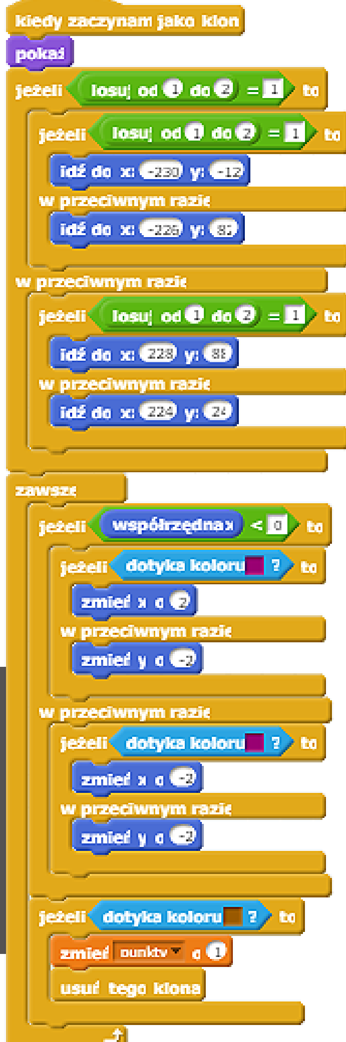


Aby stworzyć stu przeciwników w C#, stworzylibyśmy klasę **Przeciwnik**, która zawierałaby pole **kolor**. Następnie w pętli 100 razy zainicjalizowalibyśmy 100 obiektów tej klasy, nadając odpowiednie wartości polom każdego obiektu. Działanie przeciwnika opisane byłoby w klasie i byłoby wspólne dla każdego obiektu.

Podobnie wygląda tworzenie wielu gier. Programowanie obiektowe znacznie ułatwia odwzorowanie rzeczywistego świata. Wystarczy odnaleźć wspólne cechy i zachowania obiektów.

W grze wyścigowej moglibyśmy stworzyć klasę **Samochód**, w grze o piłce nożnej klasę **Piłkarz** i tak dalej. Oczywiście w jednej grze może znaleźć się bardzo dużo klas.

Jak je tworzyć, nauczymy się, realizując projekty przedstawione w kolejnym rozdziale książki.



Klony mogą wykonywać skomplikowane skrypty. W naszym przykładzie widać skrypt, którego zadaniem jest przemieszczanie każdego klona po jego utworzeniu, a następnie poruszanie nim – zależnie od tego, gdzie klon się znajduje i jakiego koloru dotyka. W tym przykładowym skrypcie widzimy, że mimo iż nie deklarujemy konkretnych pól dla klonów, każdy z nich ma swoje specyficzne właściwości, jak choćby współrzędne X i Y. Każdy klon, wykonując ten sam skrypt, potrafi dzięki temu działać inaczej

SPRAWDŹ SWOJE UMIEJĘTNOŚCI

Pod adresem **pl.spoj.com** znajdziemy polską wersję serwisu **SPOJ**. Pozwala on na sprawdzenie swoich umiejętności programistycznych poprzez rozwiązywanie zadań algorytmicznych. Zadania możemy rozwiązywać w wielu językach programowania, także w C#. Do rozwiązania zadań często nie jest wymagana duża wiedza, a raczej umiejętne zapisywanie algorytmów z wykorzystaniem najprostszych pętli i instrukcji, które właśnie poznaliśmy.



PROGRAM I PLIKI PROJEKTÓW OPISANE W TYM ROZDZIALE ZNAJDZIESZ **NA DVD I W KŚ+**

3 Zrób to sam: trzy gry w C#

Nic nie uczy tak jak praktyka. Dlatego kolejne instrukcje poznamy, tworząc w C# nasze pierwsze gry. Będzie to popularny Wisielec, rozgrywany często na kartkach, a także Memory i Snake

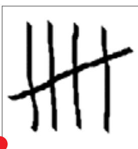
Poznaliśmy już podstawowe instrukcje języka C#, jednak aby móc z nich korzystać, należy poznać jeszcze samą strukturę kodu. Nic nie uczy tak dobrze jak praktyka. Dlatego dalszą wiedzę na temat tworzenia własnych gier w języku C# będziemy zdobywać, realizując zadania polegające na tworzeniu gier zgodnie z poradnikiem.

Bez obaw – jeśli wciąż nie czujemy się na tyle pewnie, aby uwierzyć w to, że jesteśmy już w stanie tworzyć gry. Wskazówki poprowadzą nas krok po kroku przez cały proces kodowania.

A po drodze poznamy najważniejsze zagadnienia, abyśmy kolejne gry mogli tworzyć już w pełni samodzielnie.

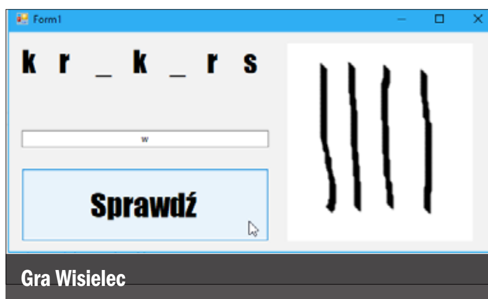
Gra 1. Wisielec

Pierwszą grą, jaką zrealizujemy, będzie Wisielec. Do tej popularnej zabawy zwykle gracze potrzebują kartki lub tablicy. My przeniesiemy ją na ekran komputera.
Gra polega na odgadywaniu liter ze słowa-zagadki. Na początku znamy tylko pierwszą i ostatnią literę słowa, a także liczbę wolnych miejsc, pozostawionych na pozostałe litery. Po każdej próbie odgadnięcia litery następuje sprawdzenie, czy znajduje się ona w słowie. Jeśli tak, jest dopisywana do słowa. W przeciwnym razie rysowany jest obrazek, zwyczajowo to szubienica. Jedna błędna litera to jedna linia na obrazku. My zamiast szubienicy zastosujemy obrazek.



Przygotowanie grafik

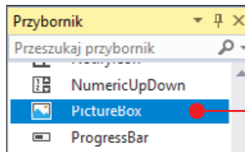
Pamiętajmy jednak, że jest to wersja końcowa obrazka. Będziemy potrzebować też wersji pośrednich. Dlatego powinniśmy stworzyć zestaw plików graficznych, zapisując krok po kroku tworzenie naszego rysunku.
Aby uzyskać przykładowy obrazek końcowy, powinniśmy mieć zapisane w oddzielnych plikach rysunki przedstawiające obrazek po dodaniu każdej kolejnej linii. Wykorzystamy w naszej grze cały zestaw grafik. Grafiki najlepiej zapisywać jako pliki **PNG**.
Oczywiście nic nie stoi na przeszkodzie, aby użyć innego zestawu pięciu grafik, by końcowy obrazek wyglądał inaczej. Wiele zależy od naszych chęci i zdolności obsługi edytorów graficznych.
Stworzenie grafik, to pierwszy etap tworzenia naszej gry. Grafiki powinny znaleźć się w repozytorium projektu. Dlatego powinniśmy utworzyć projekt. Nowy projekt tworzymy według instrukcji zaprezentowanej w pierwszym rozdziale. Możemy mu nadać nazwę **Wisielec**.



Gra Wisielec

Modelowanie okna gry

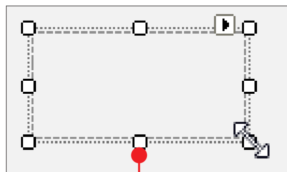
Kolejny krok to zamodelowanie okna naszej gry. Wykorzystamy do tego kontrolki znajdujące się w Przyborniku (jak to zrobić, opisano w rozdziale drugim). Potrzebnymi nam elementami będą: **PictureBox** (do pokazywania grafik po błędnej próbie), **etykiety** (kontrolki **Label**, po jednej dla każdej litery ze słowa), **TextBox** (do wpisania proponowanej litery) i **przycisk** (kontrolka **Button**, aby uruchomić sprawdzanie).

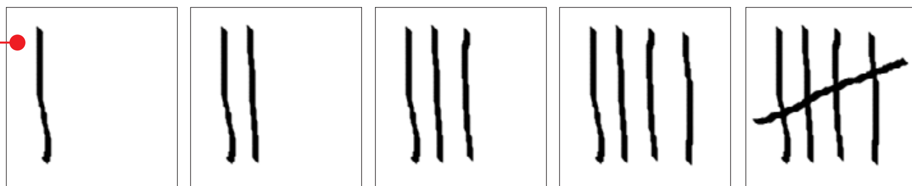


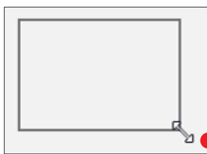
Przygotowanie miejsca na grafiki

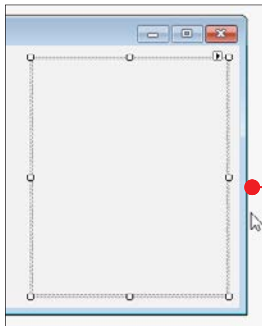
Możemy modyfikować zarówno wielkość okna gry, jak i elementów znajdujących się w tym oknie. Szczególną uwagę zwróćmy na odpowiednie dobranie wielkości kontrolki **PictureBox** – tak aby przygotowane przez nas grafiki były dobrze widoczne.

1 Aby zmienić rozmiar kontrolki, zaznaczamy ją.









2 Najeżdżamy kursorem na róg, tak aby kursor przybrał formę ukośnej strzałki z dwoma grotami.

3 Przytrzymując wciśnięty lewy przycisk myszy, przesuwamy kursor, modyfikując rozmiar kontrolki.

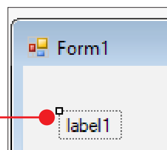
4 Powiększamy PictureBox tak, aby zajął mniej więcej połowę okna gry.

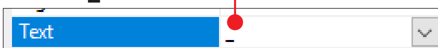
UŁATWIENIE

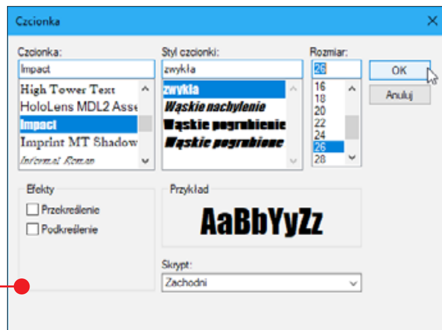
Umieszczając w oknie gry etykiety symbolizujące kolejne litery słowa-zagadki, musimy odpowiednio zmieniać właściwości w kontrolce. Aby ułatwić sobie pracę, możemy najpierw umieścić w oknie gry tylko jedną etykietę, a następnie skopiować ją kilkakrotnie, już ze zmienionymi właściwościami. Początkowo etykieta przeciągnięta z Przybornika jest dla nas widoczna jako mały obiekt z napisem. My chcemy jednak, żeby etykieta miała formę kreski, była większa, a następnie mogła przybrać formę jednej litery.

Wyświetlanie liter odgadywanego słowa

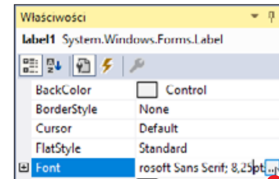
1 Zaczniemy od zmiany właściwości **Text** w etykiecie. Zamiast słowa **label1** powinno znaleźć się tam podkreślenie – _.



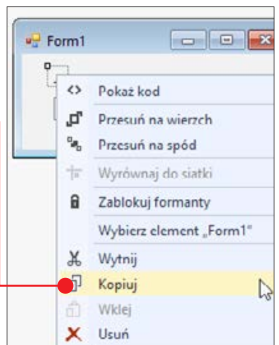




2 Kolejna modyfikacja to wielkość kontrolki. Tym razem nie zmienimy jej tak, jak w przypadku PictureBoxa – aby zwiększyć etykietę, musimy zwiększyć rozmiar czcionki, którą napisany jest znajdujący się na niej tekst. Przechodzimy zatem do właściwości **Font** – i za pomocą przycisku z wielokropkiem otwieramy okno wyboru czcionki.



3 Sformatowaną etykietę kopiujemy, a następnie wklejamy do modelu okna programu. W naszej przykładowej grze znajdzie się siedem etykiet – to oznacza, że słowa biorące udział w grze będą mogły składać się z siedmiu liter.

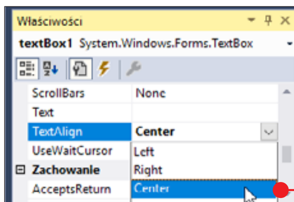




4 Ustawmy etykiety w odpowiedniej kolejności, to znaczy według nazw etykiet – od **label1** po lewej stronie do **label7** po prawej.

5 Modelując okno gry, dobierzmy także odpowiedni wygląd dla przycisku. Jego wielkość możemy zmieniać podobnie jak wielkość w PictureBoxie, a zmieniając właściwość **Font**, możemy korygować wielkość i rodzaj czcionki, jaką utworzony jest napis na przycisku. Sam napis zmożemy zmienić poprzez zmianę wartości właściwości **Text**.

6 Ostatnim elementem wymagającym zmiany jest pole tekstowe. Możemy zwiększyć jego rozmiar. Wpisywane do pola tekstowego ciągi znaków automatycznie wyświetlają się po lewej stronie pola. Atrakcyjniej byłoby, gdyby wpisywana przez gracza litera była widoczna na środku pola. W związku z tym powinniśmy zmienić dla pola tekstowego wartość właściwości **TextAlign** z **Left** na **Center**.



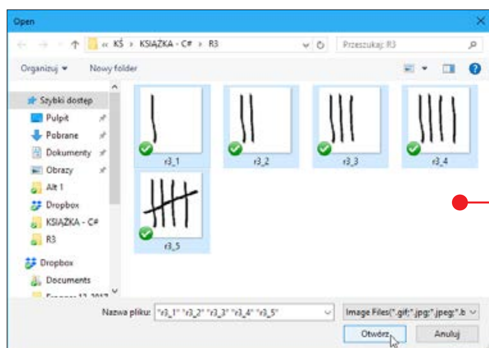
Okno programu jest już zamodelowane. Zanim przejdziemy do kodowania, mamy do skończenia jeszcze jedno zadanie. Powinniśmy dodać stworzone przez nas wcześniej grafiki do projektu, aby możliwe było ich używanie w PictureBoxie.

Import grafik do projektu

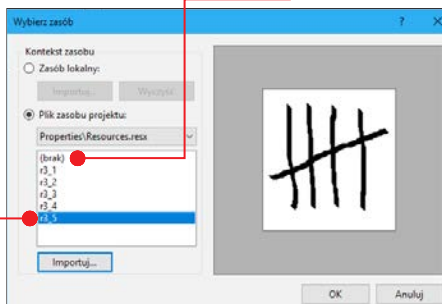
1 Przechodzimy do właściwości **Image** w PictureBoxie i klikamy na przycisk z wielokropkiem.



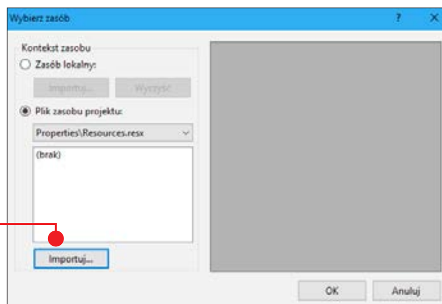
2 W nowym oknie klikamy na **Importuj**, aby otworzyć okno dialogowe pozwalające na wybranie plików graficznych. Wczytujemy wszystkie przygotowane przez nas grafiki jednocześnie.



3 Mamy jeszcze możliwość podejrzenia wgranych grafik. Jeśli pozostawimy zaznaczoną jedną z wgranych grafik – pozostanie ona widoczna w zamodelowanym przez nas oknie gry. Na początku gry nie powinno być jednak widać żadnej z grafik – dlatego zatwierdzając import i wychodząc z okna, powinniśmy zaznaczyć opcję **(brak)**.



4 Pozostaje nam do rozpatrzenia jeszcze jedna kwestia związana z PictureBoxem



– jego wielkość nie zawsze jest taka sama jak wielkość grafiki. Jeśli grafika jest większa niż pole na jej wyświetlenie, wyświetlony zostanie tylko fragment grafiki, który zmieści się w PictureBoxie (lewy górny róg zawsze jest wyświetlany). Jeśli grafika jest mniejsza niż PictureBox, zostanie wyświetlona w całości, przysunięta do lewego górnego rogu pola na grafikę. Czy to oznacza, że musimy zmieniać wielkość PictureBoxa lub naszych grafik? Oczywiście, że nie. PictureBox ma odpowiednią właściwość, której zmiana wartości pozwoli nam na odpowiednie wyświetlanie grafiki. To właściwość **SizeMode**, która może przyjmować

Form1.cs* | Form1.cs [Design]* | Object Browser
Wisielec | Wisielec.Form1

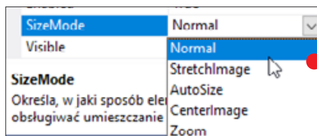
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Wisielec
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {

        }
    }
}
```

A, B, C



wartości: **Normal**, **StretchImage**, **AutoSize**, **CenterImage** i **Zoom**.

5 Należy zapoznać się z opisami, każdego ze sposobów wyświetlania grafiki, aby wybrać najlepszy dla naszego projektu. W naszym przykładzie wybieramy **Zoom** – można jednak wybrać inną opcję.

Czas na pisanie

Możemy przejść teraz do kodowania. Co ciekawe, część kodu stworzy się sama! Wy-

WARTOŚCI WŁAŚCIWOŚCI SIZEMODE

Normal	To ustawienie domyślne. Jeżeli nie zmienimy wartości właściwości samodzielnie, grafika będzie wyświetlana właśnie w ten sposób. Jeśli grafika jest większa niż pole na jej wyświetlenie, wyświetlony zostanie tylko fragment (poczynając od lewego górnego rogu) grafiki, który zmieści się w PictureBoxie. Jeżeli grafika jest mniejsza niż PictureBox, zostanie wyświetlona w całości – będzie jednak przysunięta do lewego górnego rogu pola na grafikę.
StretchImage	Niezależnie od tego, czy grafika jest mniejsza, czy większa niż pole przeznaczone na jej wyświetlenie – wymiary grafiki zostają zmienione na wymiary PictureBoxa, nie są przy tym zachowywane proporcje pomiędzy wysokością a szerokością grafiki. Prowadzi to do zdeformowania obrazu.
AutoSize	Ta wartość dopasowuje wielkość PictureBoxa do wielkości wybranej przez nas grafiki – powiększając go lub pomniejszając.
CenterImage	W przypadku grafik mniejszych niż PictureBox pokazywana jest cała grafika, która jest wyśrodkowana. Gdy grafika jest większa niż PictureBox, pokazywany jest jedynie jej środek, a obcięte zostają brzegi niemieszczące się w obszarze PictureBoxa.
Zoom	Wielkość grafiki zostaje zmieniona tak, aby cały obrazek był widoczny w PictureBoxie na możliwie dużym jego obszarze. Zachowane są przy tym proporcje grafiki.

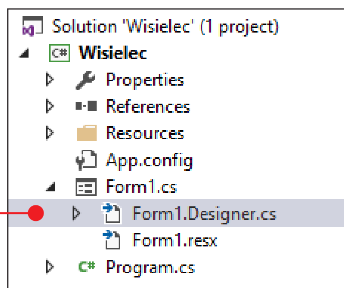
starczy, że na podglądzie okna gry klikniemy dwukrotnie na przycisk do sprawdzania poprawności udzielonej odpowiedzi. Przeniesieni zostaniemy do pliku z kodem naszego programu. Jak widać, jest tam już całkiem sporo linijek.

1 Warto dowiedzieć się, co one oznaczają. W wielu linijkach na samym początku kodu pojawia się słowo **using** **A**. Jest to bardzo przydatne narzędzie pozwalające uniknąć konieczności poprzedzania nazwą przestrzeni, typów i metod znajdujących się w danym pliku lub przestrzeni nazw. Co to dla nas oznacza? W poprzednim rozdziale poznaliśmy polecenia ***Console.ReadLine()*** i ***Console.WriteLine()***. Nie jest to jednak pełna instrukcja wywołująca te funkcje. Zwyczajowo powinniśmy napisać **System.Console.ReadLine()** i **System.Console.WriteLine()**. Dzięki dyrektywie **using System;** słowa **System** nie musimy używać już w kodzie naszej gry, odwołując się do metod z tej przestrzeni nazw. Zdecydowanie skraca to kod programu i przyspiesza pisanie.

2 Przestrzeń nazw – czyli z języka angielskiego **namespace** **B** – to słowo również pojawia się w kodzie naszego programu. A to dlatego, że sami tworząc naszą grę, wytwarzamy też pewną przestrzeń nazw, w obrębie której się poruszamy. Domyślnie nasza przestrzeń nazw nosi taką samą nazwę, jak projekt, który utworzyliśmy.

3 Widzimy, że w naszej przestrzeni nazw znajduje się jedynie klasa **Form1** **C**. Co prawda nie jest to jedyny element naszej przestrzeni nazw, ale jedyny w tym momencie dla nas widoczny i jedyny, który w tym momencie powinien nas interesować. Co więcej, jest to klasa dziedzicząca po klasie **Form**, a słowo **partial** oznacza, że jest to tylko część definicji klasy **Form1**.

4 Wiemy też, że definicja klasy **Form1** jest tylko częściowa. Co zatem z resztą – gdzie jest i co się w niej znajduje? Spójrzmy na eksplorator projektu. Po rozwinięciu pozycji **Form1.cs** widzimy pozycję **Form1.Designer.cs** – możemy otworzyć ten plik.



Tam również znajduje się częściowa definicja klasy **Form1**. Definicja ta w głównej mierze

DZIEDZICZENIE

Klasy mogą po sobie dziedziczyć – jest to jeden z fundamentów programowania obiektowego. Co to dla nas oznacza? Załóżmy, że tworzymy grę wyścigową. Dla jej potrzeb definiujemy klasę **pojazd**. Opisuje ona najważniejsze rzeczy, takie jak to, że pojazd przyspiesza, kiedy dodajemy gazu, że zwalnia, gdy hamujemy, że pojazd wymaga kierowcy. Chcemy, żeby w naszej grze można było ścigać się samochodami i motocyklami. Dlatego musielibyśmy zdefiniować też klasy **samochód** i **motocykl**. Zarówno samochód, jak i motocykl przyspiesza, kiedy dodajemy gazu, a zwalnia, gdy hamujemy. Zamiast pisać te same rzeczy w dwóch kolejnych klasach, możemy sprawić, aby klasy te dziedziczyły po klasie **pojazd**, a następnie w nich opisać tylko te rzeczy, które odróżniają samochody od motocykli. W naszym Wisielcu klasa **Form** jako klasa bazowa definiuje wszystkie ważne aspekty okna programu. Dzięki niej nie musimy martwić się choćby o to, jak napisać cały interfejs graficzny.

```
//
// button1 F
//
this.button1.Font = new System.Drawing.Font("Impact", 27.75F, System.Drawing.FontStyle.Regular, System.Drawing.Graphi
this.button1.Location = new System.Drawing.Point(15, 162);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(294, 87); F
this.button1.TabIndex = 9;
this.button1.Text = "Sprawdź"; F
this.button1.UseVisualStyleBackColor = true;
this.button1.Click += new System.EventHandler(this.button1_Click);
```

opiera się na metodzie ***InitializeComponent()*** **D** – jej treść generowała się, gdy modelowaliśmy okno gry. Linijka **this.pictureBox1 = new System.Windows.Forms.PictureBox();** **E** powstała, gdy przeciągnęliśmy PictureBox z przybornika. Podobnie jest z kolejnymi linijkami – powstawały one, gdy dodawaliśmy do gry kolejne kontrolki.

5 Popatrzmy na ten kod nieco dalej. Znajdziemy tam opisy każdej dodanej przez nas kontrolki. W naszej grze znajduje się przycisk, nie wygląda on jednak tak, jak w momencie przeciągnięcia go z przybornika. Zmienialiśmy jego właściwości. To również jest opisane w kodzie klasy **Form1**. Znajdziemy tam fragment metody ***InitializeComponet()***, który zmienia właściwości przycisku według podanych przez nas parametrów. Możemy zobaczyć, jak zmienia się właściwość **Font**, **Size** czy **Text** **F**. Jeśli wrócimy do naszej wcześniej omawianej części klasy **Form1** – również zobaczymy metodę ***InitializeComponent();*** ●, tyle że nie jej definicję, ale jej wywołanie.

```
Form1.cs*   Form1.cs [Design]*   Object Browser
Wisielec                          Wisielec.Form1

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Wisielec
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent(); ●
        }
```

```
Form1.Designer.cs   Program.cs   Form1.cs   Form1.cs [Design]   Object Browser
Wisielec                          Wisielec.Form1

namespace Wisielec
{
    partial class Form1
    {
        /// <summary> Required designer variable.
        private System.ComponentModel.IContainer components = null;

        /// <summary> Clean up any resources being used.
        protected override void Dispose(bool disposing)...

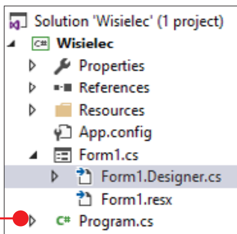
        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent() D
        {
            this.pictureBox1 = new System.Windows.Forms.PictureBox(); E
            this.label1 = new System.Windows.Forms.Label();
            this.label2 = new System.Windows.Forms.Label();
            this.label3 = new System.Windows.Forms.Label();
            this.label4 = new System.Windows.Forms.Label();
            this.label5 = new System.Windows.Forms.Label();
            this.label6 = new System.Windows.Forms.Label();
            this.label7 = new System.Windows.Forms.Label();
            this.textBox1 = new System.Windows.Forms.TextBox();
            this.button1 = new System.Windows.Forms.Button();
```

Konstruktor

Aby można było skorzystać z danej metody, musi ona zostać wywołana. W naszej klasie jest ona wywoływana bezpośrednio w konstruktorze. **Konstruktor** to taka specyficzna metoda, której wywołaniu towarzyszy utworzenie obiektu klasy. Wywołując konstruktor klasy **Form1**, tworzymy nasze

okno programu, a poprzez metodę ***InitializeComponent()*** umieszczamy na nim wszystkie elementy wybrane przez nas podczas modelowania.
Jeśli konstruktor jest metodą, to żeby móc z niego skorzystać, powinien zostać wywołany. Pytaniem pozostaje – kiedy to się dzieje? Spójrzmy jeszcze raz na eksplorator projektu. Znajduje się w nim też plik **Program.cs** ●. Zapoznajmy się z jego treścią. Mamy tam kolejny kod, definiujący klasę **Program**, która również jest w przestrzeni nazw **Wisielec**. Klasa **Program** ma tylko jedną metodę – ***Main()*** G. Jest to główna i najważniejsza metoda w naszym projekcie, ponieważ jest ona wywoływana w momencie startu programu. Jeśli przyjrzymy się treści tej metody, zobaczymy, że tam wywoływany jest konstruktor klasy **Form1** H. Z metodą ***Main()*** możemy spotkać się też w innych językach, jak choćby C++ czy Java.



KOMENTARZE

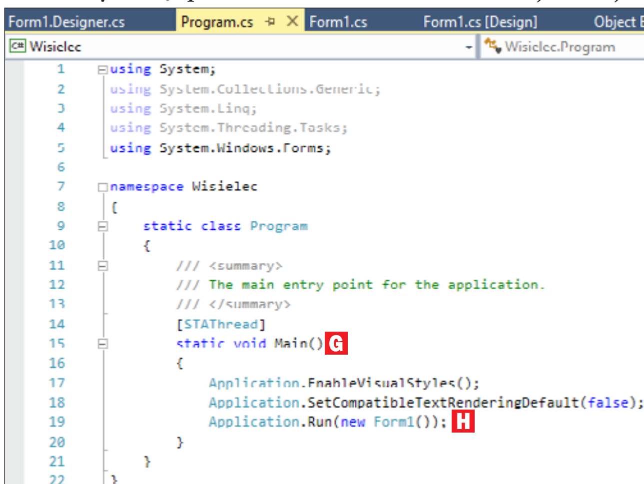
Linie kodu poprzedzone dwoma ukośnikami – // ● bądź zawarte pomiędzy /* a */ to komentarze.

```
//
// button1
//
this.button1.Font = new System.Drawi
this.button1.Location = new System.D
this.button1.Name = "button1";
```

Komentarze to fragmenty kodu, które nie są brane pod uwagę podczas kompilowania. W treści komentarza nie musimy trzymać się zasad języka, ponieważ komentarze są widoczne tylko dla osoby widzącej kod, a dla komputera nie są istotne.
Programiści stosują komentarze, aby sami mogli lepiej orientować się w tworzonym kodzie.
Nad dużymi projektami pracuje wiele osób – warto wtedy opisywać poszczególne klasy i zawarte w nich metody komentarzami, aby współpracownicy wiedzieli, do czego służą.

Tworzymy nowe metody

Kiedy zapoznaliśmy się już z podstawową strukturą kodu i dowiedzieliśmy się, skąd wziął się automatycznie wygenerowany kod programu, możemy zacząć pisać samodzielnie.



```
Form1.Designer.cs   Program.cs   Form1.cs   Form1.cs [Design]   Object E
Wisielec                          Wisielec.Program

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Wisielec
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main() G
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1()); H
        }
    }
}
```

```
private void button1_Click(object sender, EventArgs e)
{

}
```

1 W naszej klasie **Form1** wygenerowała się jeszcze jedna metoda. Miało to miejsce, gdy klikaliśmy dwukrotnie na przycisk na modelowanym oknie gry. Metoda ***button1_Click()*** ● będzie wywoływana zawsze, gdy gracz kliknie na dany przycisk. W metodzie tej nie ma nic wewnątrz nawiasu klamrowego. Dlatego nie ma ona jeszcze żadnej treści. Naszym zadaniem będzie właśnie nadanie jej treści. Zastanówmy się: co powinno się dziać po kliknięciu na przycisk? Otóż powinno

```
private void losuj_slowo()
{
    string slowo; C
    string[] slowa = { "krokusy", "liliput", "marchew", "selerek", "krakers", "klakier" };
}
```

NAZWA METODY

Dobierając nazwę dla metody, musimy użyć jednego ciągu znaków. Znak spacji oznacza koniec nazwy polecenia. Tylko że nazwa metody powinna wskazywać na to, do czego służy metoda, co często wymaga kilku słów. Aby nie oddzielać ich znakiem spacji, stosuje się znak _

```
private void losuj_slowo()
{

}
```

wtedy nastąpić sprawdzenie, czy wprowadzona litera znajduje się w słowie-zagadce. Ale żeby móc wykonywać jakiekolwiek sprawdzenie litery, powinniśmy najpierw wybrać słowo-zagadkę. Wybór nastąpi poprzez jego wylosowanie.

2 Proces losowania słowa-zagadki umieścimy w oddzielnej metodzie. Jej definicję powinniśmy zaś umieścić wewnątrz klasy **Form1**, ale poza definicjami innych metod. Najlepiej zacząć pisanie po zamknięciu nawiasu klamrowego od metody ***button1_Click()***. Stworzymy metodę ***losuj_slowo()***. Przed nazwą metody powinniśmy napisać jeszcze

METODY O TYPIE VOID

Z zasady: metody powinny zwracać pewne dane i z tego, co zwraca metoda, można korzystać jak z wartości zmiennej. Tylko że, jak wiadomo, zasady są po to, aby je łamać – są metody, które nie zwracają danych w żadnym typie i możemy je traktować po prostu jako wyodrębnione fragmenty kodu. Są to metody o typie **void** – nasza metoda nie będzie zwracała danych w żadnym typie. Będzie ona właśnie takim wyodrębnionym fragmentem kodu.

private – co oznacza, że metoda będzie dostępna tylko w naszej klasie, a także **void** – co oznacza typ metody.

TYPY ZMIENNYCH

Do stworzenia dobrego kodu będziemy potrzebować kilku zmiennych, a deklarując je, będziemy podawać ich typ. Zmienna składa się z nazwy i wartości. Wartość zmiennej może być różna. Jak jej nazwa wskazuje, może się zmieniać – ale nie bezgranicznie, ponieważ musi się ona mieścić w pewnym zakresie. O zakresie wartości mówi typ zmiennej. Ważnym typem zmiennej jest **string** A – czyli ciąg znaków. Tego typu danych używamy do przechowywania tekstów. Innym ważnym typem zmiennej jest **Integer** – w skrócie **int** B. Służy do przechowywania liczb, ale jedynie liczb całkowitych (aby przechowywać ułamki, możemy wykorzystać typy danych **float** lub **double**).

3 Teraz, zanim przystąpimy do definiowania treści tej metody, powinniśmy dowiedzieć się, czym są **zmienne**, a szczególnie ich typy. Jest ich sporo, ale na potrzeby naszej gry wystarczy nam poznanie tylko kilku. Ponieważ słowo-zagadka jest tekstem, będziemy potrzebować zmiennej typu **string**. Deklarując zmienną, podajemy jej typ oraz nazwę, następnie po znaku równości możemy podać wartość zmiennej. **Teksty umieszczamy zawsze w cudzysłowie. Pamiętajmy też o zakończeniu linii kodu średnikiem.**

4 Nie tylko zmienne przechowują dane, robią to również **tablice**. Będziemy po-

```
private void losuj_slowo()
{
    string slowo = "marchew";
}
```

```
private void losuj_slowo()
{
  A string slowo;
    string[] slowa = { "krokusy", "liliput", "marchew", "selerek", "krakers", "klakier" };
  B int ile_slow = slowa.Length;
}        D           E
```

TABLICE

To zbiory uporządkowanych danych tego samego typu, których elementy dostępne są za pomocą kluczy (indeksu). W uproszczeniu: tablica pozwala pod jedną nazwą przechować wiele wartości.

trzebować takiej tablicy. Chcemy, aby słowo-zagadka w grze było losowane. Powinniśmy zatem mieć pewien zestaw słów, z których jedno zostanie wylosowane. Zestaw tych słów powinniśmy zapisać właśnie w tablicy. Tworząc tablicę, również określamy typ przechowywanych przez nią danych. Stworzymy zatem tablicę typu **string** i przypiszemy do niej zestaw słów. Aby gra była ciekawsza, zestaw słów powinien być możliwie duży. Możemy śmiało dopisać znacznie więcej słów do tablicy. Ważne, żeby słowa te miały siedem znaków, ponieważ przygotowaliśmy okno gry na siedmioliterowe słowo. Zmienna **slowo** swoją wartość dostanie spośród zestawu wartości z tablicy – dlatego teraz nie musimy przypisywać jej żadnej wartości C.

5 Zajmiemy się teraz losowaniem. Aby odnieść się do konkretnej wartości elementu z tablicy, musimy użyć indeksu – czyli numeru. Dodając wartości do tablicy, umieściliśmy je w konkretnej kolejności. Pierwszy element tablicy ma indeks **0**. Aby się do niego odnieść, korzystamy z nazwy tablicy, a następnie w kwadratowym nawiasie wpisujemy indeks. Dla naszego przykładu użycie **slowa[0]** będzie oznaczało odniesienie się do wartości **krokusy**, natomiast **slowa[4]** będzie oznaczało odniesienie się do wartości **krakers**. Aby przypisać teraz jedną z wartości z tablicy do zmiennej **slowo**, powinniśmy wylosować indeks tej wartości. Żeby przystąpić do losowania, powinniśmy znać zakres losowania. Najmniejszym możliwym do wylosowania indeksem będzie **0**. Jaki będzie największy? Będzie on o jeden mniejszy od liczby wszystkich słów wpisanych do tablicy. Powinniśmy zatem znać liczbę słów wpisanych do tablicy – jeśli chcemy stworzyć rozbudowaną grę, słów może być kilkadziesiąt czy kilkaset. Zamiast liczyć je ręcznie, możemy zliczyć je automatycznie – unikniemy wtedy możliwości pomyłki. Ponieważ liczba słów jest liczbą całkowitą, stworzymy zmienną **ile_slow** D w typie **Integer**. Następnie nadamy jej wartość będącą liczbą elementów w tablicy **slowa**. Robimy to poprzez polecenie **Length** E poprzedzone nazwą tablicy i kropką.

6 Wiemy już, jaka będzie górna granica losowania – możemy teraz zacząć losowanie. Aby to zrobić, będziemy potrzebować generatora losowania. Poprzez linię **Random gen = new Random();** – tworzymy obiekt

```
Random gen = new Random();
int indeks_slowa = gen.Next(0, ile_slow);
```

klasy **Random**, dzięki któremu dostaniemy dostęp do metod pozwalających na uzyskanie liczb pseudolosowych.

7 Dla generatora możemy skorzystać z metody, która da nam wartość losową. To metoda **Next**, która jako parametr przyjmuje granice losowania. Jak zauważyliśmy, wywołując niektóre metody, stawiamy pusty nawias. Tym razem jednak do nawiasu wpisujemy parametry metody. Są to wartości, które przekazujemy metodzie, aby mogła dobrze zadziałać. Zgodnie z tym, co ustaliliśmy wcześniej, w nawiasie podajemy granice losowania: dolną **(0)** i górną (liczba słów w tablicy – jeśli słów jest 6, będzie to o jeden mniej niż 6, czyli 5). Metoda **Next** zwróci nam liczbę losową z podanego zakresu. Liczba ta będzie liczbą całkowitą, dlatego zapiszemy ją do odpowiedniej zmiennej w typie **int**.

```
private void losuj_slowo()
{
    string slowo;
    string[] slowa = { "krokusy", "liliput", "marchew", "selerek", "krakers", "klakier" };
    int ile_slow = slowa.Length;
    Random gen = new Random();
    int indeks_slowa = gen.Next(0, ile_slow);
    slowo = slowa[indeks_slowa];
    label1.Text = Convert.ToString(slowo[0]);
    label7.Text = Convert.ToString(slowo[6]);
}
```

8 Dalej możemy przypisać do zmiennej **slowo** wartość z tablicy **slowa** o wylosowanym indeksie.

9 Kiedy słowo jest już wybrane, możemy zrobić jeszcze jedną, dość ważną dla rozgrywki rzecz. Aby gracz mógł sprawnie odgadywać słowo, powinien znać pierwszą i ostatnią literę. Dlatego etykiety pierwsza i ostatnia powinny wyświetlać pierwszą i ostatnią literę wylosowanego słowa.
Co ciekawe, zmienna typu **string** jest tak naprawdę tablicą danych typu **char** (czyli jeden znak). Podając nazwę zmiennej typu **string**, a następnie w nawiasie kwadratowym indeks, możemy odnieść się do konkretnego znaku z ciągu znaków, czyli do konkretnej litery słowa.
Nas interesują litery o indeksie **0** (pierwsza) i **6** (ostatnia). Pierwszą literę ze słowa powinniśmy przypisać do właściwości **Text** w etykiecie **label1**, a ostatnią literę ze słowa, jako wartość właściwości **Text** w etykiecie **label7**. Ponieważ właściwość **Text** musi mieć wartość w typie **string**, nie możemy napisać **label1.Text = slowo[0];** – a to dlatego, że **slowo[0]** ma już typ **char**. Musimy zatem przekonwertować literę na typ **string** i dopiero potem dokonać przypisania.

10 Aby zmienna **slowo** była dostępna nie tylko wewnątrz metody ***losuj_slowo()***, powinna być zadeklarowana nie w metodzie, ale jako pole klasy. Wtedy dostęp do jej wartości uzyskają też inne metody klasy **Form1** – zależy nam na tym, ponieważ po kliknięciu na przycisk program powinien znać słowo, aby sprawdzać, czy znajduje się w nim podana litera. Usuwamy deklarację zmiennej z metody i umieszczamy ją u samej góry klasy **Form1**.

```
public partial class Form1 : Form
{
    String slowo;
```

11 Metoda ***losuj_slowo()*** powinna zostać wywołana, aby słowo zostało wylosowane. Najlepiej wywołać ją zaraz po uruchomieniu gry. Wywołanie metody umieścimy w konstruktorze.

```
public Form1()
{
    InitializeComponent();
    losuj_slowo();
}
```

Kliknięcie na przycisk

Teraz możemy przejść do pisania kodu, który wykona się po kliknięciu na przycisk. Przenosimy się zatem do wnętrza metody ***button1_Click***. Zastanówmy się teraz, jakie informacje już mamy i co powinniśmy uzyskać.

1 Po wpisaniu przez gracza litery w pole tekstowe możemy ją odczytać i zapisać do odpowiedniej zmiennej. Będzie to nam potrzebne do porównywania tej litery z ko-

```
private void losuj_slowo()
{
    string[] slowa = { "krokusy", "liliput", "marchew", "selerek", "krakers", "klakier" };
    int ile_slow = slowa.Length;
    Random gen = new Random();
    int indeks_slowa = gen.Next(0, ile_slow);
    slowo = slowa[indeks_slowa];
    label1.Text = Convert.ToString(slowo[0]);
    label7.Text = Convert.ToString(slowo[6]);
}
```

lejnymi literami słowa-zagadki. Deklarujemy zatem zmienną **litera** i przypisujemy jej jako wartość to, co znajduje się w polu tekstowym (**textBox1.Text**).

```
private void button1_Click(object sender, EventArgs e)
{
    string litera = textBox1.Text;
    bool czy_trafiony = false;
    int gdzie_trafiony = 0;
```

2 Nie jest to jedyna zmienna, której będziemy potrzebować do stworzenia skryptu. Musimy też gdzieś przechować informację o tym, czy podana litera jest prawidłowa i należy ją wyświetlić, czy też nie jest i trzeba wyświetlić obrazek oznaczający, że gracz spudłował. Do tego również użyjemy zmiennej. Moglibyśmy użyć zmiennej tekstowej typu **string** i nadawać jej wartość **tak** lub **nie**. Nie jest to jednak najlepsze rozwiązanie. Do tego typu zadań służy specjalny typ zmiennych. Są to zmienne boolowskie (logiczne) – typ **bool**. Mogą one przybierać jedynie wartości **true** i **false**, czyli prawda i fałsz. Ze zmiennych tego typu korzystamy zazwyczaj, kiedy zmienna odpowiada na pytanie **czy...?**. Dlatego naszą zmienną typu **bool** możemy w tym wypadku nazwać **czy_trafiony**. Jej początkowa wartość to **false**, dlatego że przed sprawdzeniem nie możemy odpowiedzieć twierdząco na pytanie, czy litera jest prawidłowa. Będzie to możliwe dopiero po sprawdzeniu.

3 Informacja o tym, czy litera została odgadnięta, to jednak zbyt mało. Nas interesuje jeszcze to, na którym miejscu w słowie znajduje się trafiona litera. Musimy to wiedzieć, aby wpisać literę na odpowiedniej etykiecie. Tworzymy zatem kolejną zmienną, której wartość będzie liczbą całkowitą – dlatego wybieramy typ **integer**. Nazwa zmiennej będzie wskazywać na to, że przechowuje ona numer trafionej litery. Możemy zmiennej nadać wartość początkową równą **0**.

4 Sprawdzanie, czy litera została trafiona, będzie wyglądało tak samo dla każdej litery, która znajduje się w słowie. Aby wielokrotnie wykonać podobną czynność, możemy użyć pętli. Wiemy, ile razy pętla powinna się wykonać – słowo ma określoną liczbę liter. A w takiej sytuacji (kiedy wiemy, ile razy pętla powinna się wykonać) należy zastosować pętlę **for**. Nasza pętla powinna właściwie wykonać się jedynie pięć razy – ponieważ słowo ma siedem liter, a pierwsza i ostatnia litera są już wpisane do okna gry. Litery w słowie ponumerowane są od 0 do 6 – dlatego powinniśmy sprawdzić jedynie litery o numerach **1**, **2**, **3**, **4** i **5**. Nasza pętla może iterować na zmiennej właśnie w takim zakresie. Dlatego tworząc pętlę **for**, deklarujemy zmienną **i** typu **int** o wartości początkowej **1**, która będzie rosła o jeden (**i++**), dopóki będzie mniejsza niż **6**.

```
for (int i = 1; i < 6; i++)
{

}
```

5 Kontynuujemy pisanie wewnątrz pętli **for**. Skorzystamy teraz z instrukcji warunkowej (**if**), aby sprawdzić, czy podana przez gracza litera jest taka sama jak kolejne litery ze zmiennej **slowo** rozpatrywanej jako tablica znaków. Ponieważ odnosimy się do poszczególnych znaków, które są rozumiane jako zmienne innego typu (**char**), musimy je konwertować do typu **string**,

```
for (int i = 1; i < 6; i++)
{
    if (Convert.ToString(slowo[i]) == litera)
    {
        czy_trafiony = true;
        gdzie_trafiony = i;
    }
```

w jakim przechowywana jest podana przez gracza litera. Jeśli podana litera znajduje się w słowie, nadajemy zmiennym **czy_trafiony** i **gdzie_trafiony** odpowiednie wartości (odpowiednio **true** i **indeks** litery).

6 Pozostajemy nadal w pętli **for**. Kiedy wiemy już, na którym miejscu w słowie znajduje się trafiona litera, możemy wpisać ją na etykietę. W tym celu musimy odczytać,

jaka jest wartość zmiennej **gdzie_trafiony** i zależnie od niej wypisać literę na odpowiedniej etykiecie. Do tego znów wykorzystamy instrukcje warunkowe. Musimy po kolei od **1** do **5** sprawdzać, jaką wartość ma zmienna – jeżeli **1**, to wypisujemy literę na etykiecie **label2**, jeśli **2** to na etykiecie **label3** itd. Wypisanie litery na etykiecie ogranicza się do przypisania wartości zmiennej do właściwości **Text** etykiety.

```
if (gdzie_trafiony == 1) { label2.Text = litera; }
if (gdzie_trafiony == 2) { label3.Text = litera; }
if (gdzie_trafiony == 3) { label4.Text = litera; }
if (gdzie_trafiony == 4) { label5.Text = litera; }
if (gdzie_trafiony == 5) { label6.Text = litera; }
```

7 Po wyjściu z pętli **for** powinniśmy zająć się przypadkiem, w którym zmienna **czy_trafiony** po przejściu wszystkich iteracji pętli nie zmieniła swojej wartości na **true**. Oznacza to, że gracz nie trafił prawidłowej litery i należy wyświetlić obrazek mówiący o liczbie pudeł. Zaczynamy od stworzenia instrukcji warunkowej, aby sprawdzić, czy zmienna **czy_trafiony** ma wartość **false**.

```
if (czy_trafiony == false)
{

}
```

8 Każde kliknięcie na przycisk sprawdzający wywołuje pętlę **for** i może skutkować doliczeniem kolejnego pudła. Zmienna przechowująca liczbę pudeł nie może być zatem tworzona wewnątrz metody ***button1_Click***, ponieważ jej wartość z każdym kliknięciem mogłaby się zerować. Wartość tej zmiennej powinna być przechowywana poza metodą – dlatego zmienną przechowującą liczbę pudeł zadeklarujemy jako pole klasy

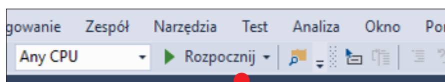
```
public partial class Form1 : Form
{
    String slowo;
    int ile_pudel = 0;
```

– podobnie jak zmienną przechowującą słowo-zagadkę. Na samym początku gry pudeł powinno być oczywiście **0**.

```
if (czy_trafiony == false)
{
    ile_pudel = ile_pudel + 1;
}
```

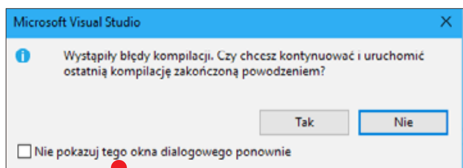
9 Liczba pudeł powinna zwiększać się, kiedy zmienna **czy_trafiony** po przejściu pętli **for** zachowa wartość **false**.

10 Ponieważ wraz ze zwiększeniem wartości zmiennej wiemy, że gracz ma już na koncie jakieś pudła, możemy wyświetlić jeden z obrazków wgranych wcześniej przez nas do PictureBoxa. Aby określić, który obrazek wyświetlimy, z pomocą instrukcji warunkowych musimy określić, jaka jest wartość pola **ile_pudel**. Instrukcji będzie tyle, ile przygotowaliśmy obrazków do wyświetlenia. W naszym przykładzie mamy ich pięć, dlatego sprawdzamy, czy **ile_pudel** przyjmuje którąś z wartości od **1** do **5** i na tej podstawie wyświetlany jest odpowiedni obrazek. Instrukcja **pictureBox1.Image = Wisielec.Properties.Resources.r3_1** pozwala na przypisanie w naszym PictureBoxie obrazka, który znajduje się w zasobach projektu. Słowo **Wisielec** w instrukcji jest nazwą projektu (jeśli nazwaliśmy projekt inaczej, musimy użyć innego słowa). **Properties.Resources** wskazuje na to, gdzie przechowywane są zasoby, natomiast **r3_1** – to nazwa grafiki. Tu również powinniśmy napisać nazwę, której użyliśmy.

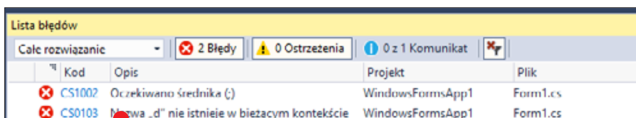


Można powiedzieć, że zbudowana w ten sposób gra już działa. Aby ją przetestować, klikamy na przycisk **Rozpocznij**. Jeśli coś jest nie tak z kodem programu – Visual Studio nie uruchomi naszej gry, wyświetlając następują-

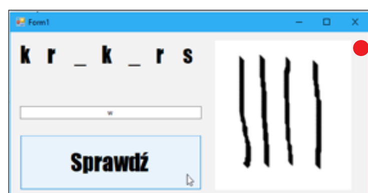
```
if (czy_trafiony == false)
{
    ile_pudel = ile_pudel + 1;
    if (ile_pudel == 1) { pictureBox1.Image = Wisielec.Properties.Resources.r3_1; }
    if (ile_pudel == 2) { pictureBox1.Image = Wisielec.Properties.Resources.r3_2; }
    if (ile_pudel == 3) { pictureBox1.Image = Wisielec.Properties.Resources.r3_3; }
    if (ile_pudel == 4) { pictureBox1.Image = Wisielec.Properties.Resources.r3_4; }
    if (ile_pudel == 5) { pictureBox1.Image = Wisielec.Properties.Resources.r3_5; }
}
```



ce okno ●. Po zaniechaniu próby uruchomienia u dołu ekranu powinna pokazać się lista



błędów ●, które uniemożliwiły włączenie gry. Program pokazuje nam, jaki błąd został wykryty, a także linię, w której się on znajduje. Podczas pisania częstymi błędami są literówki, a także brak średnika na końcu polecenia. Kiedy poprawimy już ewentualne błędy, powinno pojawić się okno naszej gry ● – możemy ją przetestować.



Testowanie

Podczas testów zapewne okaże się, że jeśli nasza gra jest stworzona w opisany sposób, to niezależnie od tego, czy odgadliśmy słowo, czy nie – nie kończy się. Powinniśmy znaleźć sposób, jak pokazać graczowi, że wygrał, a także jak uniemożliwić mu dalsze odgadywanie zarówno, gdy wygrał, jak i po wyświetleniu ostatniego obrazka, oznaczającego przegraną.

Dopracowywanie wykrytych niedociągnięć

Nasza gra będzie wyświetlała grafikę **Wygrywasz** po odgadnięciu słowa



i grafikę **Gra skończona** ●, kiedy gracz pomimo wyświetlenia już ostatniego obrazka, wyczerpującego liczbę prób, będzie dalej odgadywał hasło.

1 Przechodzimy do właściwości **Image** naszego PictureBoxa i podobnie jak wcześniej wgrywamy do niego kolejne pliki graficzne – jeden do pokazywania po wygranej, a drugi po przegranej.

2 Tworzymy w naszej klasie **Form1** metodę ***wygrana()***. Będzie ona wywoływana po kliknięciu na przycisk i sprawdzi, czy nastąpiła już wygrana – jeśli tak, wyświetlony zostanie stosowny obrazek. A czym jest wygrana w kontekście naszego kodu?
To moment, kiedy wszystkie etykiety mają we właściwości **Text** wartość inną niż „_". Sprawdzamy, czy etykieta **label2** ma wartość właściwości **Text** inną niż „_" – zapis „**!=**" ● oznacza **inne niż**. Jeśli jest to prawda, przechodzimy do sprawdzania następnej etykiety – i tak dalej, umieszczając kolejne instrukcje warunkowe wewnątrz poprzednich. Sprawdzanie wykonujemy tylko na pięciu etykietach, ponieważ pierwsza i ostatnia zawsze mają wpisane litery. Kiedy **Text** we wszystkich sprawdzanych etykietach ma wartość inną niż „_", możemy wyświetlić w PictureBoxie obrazek mówiący o wygranej ● – w taki sam sposób, jak wy-

```
private void wygrana()
{
        if (label2.Text != "_")
        {
            if (label3.Text != "_")
            {
                if (label4.Text != "_")
                {
                    if (label5.Text != "_")
                    {
                        if (label6.Text != "_")
                        {
                        pictureBox1.Image = Wisielec.Properties.Resources.wygrana;
                        }
                    }
                }
            }
        }
}
```

świetlaliśmy inne grafiki. Na końcu metody ***button1_Click()*** wywołujemy metodę ***wygrana()***.

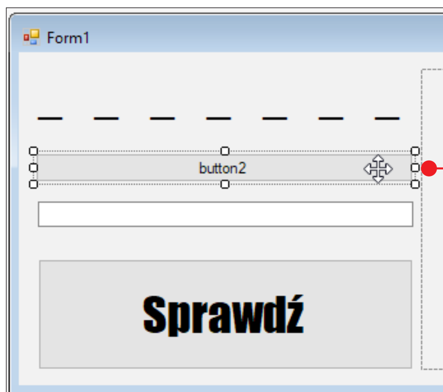
3 Powinniśmy zrealizować jeszcze wyświetlanie grafiki **Gra skończona** – gdy gracz próbuje dalej odgadywać, mimo że wyczerpał limit prób. Cały kod, który napisaliśmy wewnątrz metody ***button1_Click()***, powinien wykonywać się tylko wtedy, gdy liczba pudeł jest mniejsza niż 5. Dlatego na początku metody dodajemy instrukcję **if** sprawdzającą, ile pudeł ma na koncie gracz. Jeśli jest ich mniej niż 5, będzie wykonywał się kod napisany przez nas wcześniej. Po napisanym przez nas warunku otwieramy

```
private void button1_Click(object sender, EventArgs e)
{
    if (ile_pudel < 5)
    {
        string litera = textBox1.Text;
```

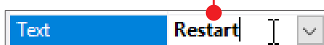
nawias klamrowy, zamykamy go natomiast na samym końcu metody, już po wywołaniu metody ***wygrana()*** **A**. Kiedy liczba pudeł nie jest już mniejsza niż 5, powinno nastąpić wyświetlenie grafiki. Dlatego na naszej instrukcji warunkowej dopisujemy na końcu metody sekcję **else** **B**, która wykona podany wewnątrz nawiasu klamrowego kod, gdy warunek określony na samym początku metody nie zostanie spełniony. W tym wypadku powinno nastąpić wyświetlenie grafiki w PictureBoxie za pomocą znanego już polecenia **C**.

Restart gry

Nasza gra pozwala na przeprowadzenie tylko jednej rozgrywki. Aby zagrać ponownie, należy uruchomić grę po raz kolejny. Możemy jednak zmienić grę w taki sposób, aby pozwalała na więcej niż jedną rozgrywkę. Potrzebny nam będzie do tego kolejny przycisk.

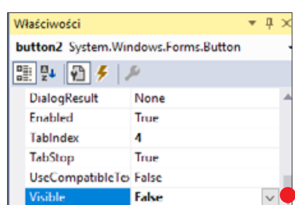


Musimy powrócić do modelowania okna gry (patrz strona 15) i umieścić na oknie drugi przycisk, któremu zmieniamy właściwość **Text** na **Restart**.



Rozpoczęcie nowej gry powinno być możliwe dopiero po zakończeniu poprzedniej rozgrywki. Przycisk do restartu powinien być zatem niewidoczny aż do momentu wyświetlenia grafiki oznaczającej wygraną lub przegraną.

1 Aby przycisk był niewidoczny po uruchomieniu gry, zmieniamy wartość jego właściwości **Visible** na **False**.



2 Z poziomu kodu możemy teraz sprawić, aby przycisk znów stał się widoczny. Musimy to zrobić w dwóch miejscach. Pierwsze to wnętrze metody ***wygrana()*** – po tym, gdy wyświetlimy grafikę wygranej. Ponowne po-

```
            if (ile_pudel == 4) { pictureBox1.Image = Wisielec.Properties.Resources.r3_4; }
            if (ile_pudel == 5) { pictureBox1.Image = Wisielec.Properties.Resources.r3_5; }
        }
        wygrana(); A
    }
    else B
    {
        pictureBox1.Image = Wisielec.Properties.Resources.przegrana; C
    }
}
```

```
if (czy_trafiony == false)
{
    ile_pudel = ile_pudel + 1;
    if (ile_pudel == 1) { pictureBox1.Image = Wisielec.Properties.Resources.r3_1; }
    if (ile_pudel == 2) { pictureBox1.Image = Wisielec.Properties.Resources.r3_2; }
    if (ile_pudel == 3) { pictureBox1.Image = Wisielec.Properties.Resources.r3_3; }
    if (ile_pudel == 4) { pictureBox1.Image = Wisielec.Properties.Resources.r3_4; }
    if (ile_pudel == 5) {
        pictureBox1.Image = Wisielec.Properties.Resources.r3_5;
        button2.Visible = true;
    }
}
```

kazanie przycisku uzyskamy poprzez przypisanie do jego właściwości **Visible** wartości **true**.

```
if (label6.Text != "_")
{
pictureBox1.Image = Wisielec.Properties.Resources.wygrana;
button2.Visible = true;
}
```

3 Tę samą linijkę kodu powinniśmy umieścić jeszcze w jednym miejscu – po wyświetleniu ostatniej grafiki, obrazującej piąte pudło gracza.

4 Pozostało nam jeszcze do zaprogramowania zareagowanie na kliknięcie na nowy przycisk. Wracamy do modelowania okna gry i dwukrotnie klikamy na przycisk do restartu. Program przenosi nas z powrotem do naszej klasy **Form1**, jednak jest już w niej wygenerowana metoda ***button2_Click()***,

```
private void button2_Click(object sender, EventArgs e)
{

}
```

która wywoływana będzie po kliknięciu na omawiany przycisk. Metoda ta oczywiście nie zawiera jeszcze treści. Musimy ją napisać.

5 Czym jest restart gry? Sprowadza się on do ponownego wywołania losowania słowa – a to uzyskamy, wywołując metodę ***losuj_slowo()*** **A**, wyzerowania liczby pudeł **B** i zmiany tekstu w etykietach **C**, aby znów były tam podkreślenia oznaczające puste miejsca na litery.

```
private void button2_Click(object sender, EventArgs e)
{
    losuj_slowo(); A
    label2.Text = "_";
    label3.Text = "_";
    label4.Text = "_";
    label5.Text = "_";
    label6.Text = "_";   C
    ile_pudel = 0; B
}
```

6 By ponownie zablokować możliwość restartu gry, przycisk po kliknięciu na niego powinien się ukryć. Pod koniec metody ***button2_Click()*** możemy napisać przypisanie wartości **false** do właściwości **Visible**

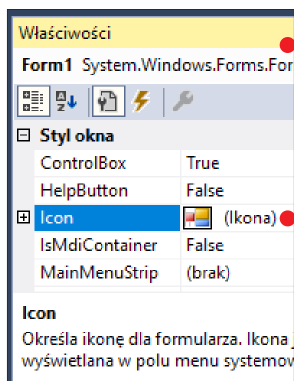
```
private void button2_Click(objec
{
    losuj_slowo();
    label2.Text = "_";
    label3.Text = "_";
    label4.Text = "_";
    label5.Text = "_";
    label6.Text = "_";
    ile_pudel = 0;
    button2.Visible = false;
    pictureBox1.Image = null;
}
```

w przycisku. A żeby przywrócić **PictureBox** do stanu początkowego, zmieniamy wartość jego właściwości **Image**, przypisując mu puste pole, które widoczne było na początku gry.

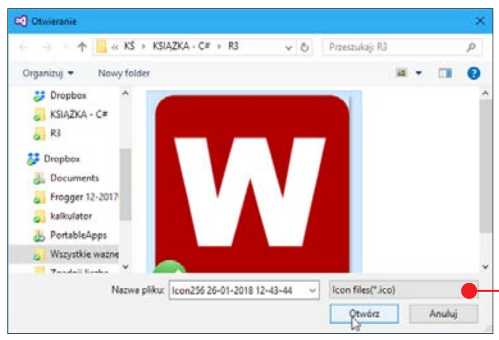
7 Nasza gra działa już całkiem sprawnie. Wykonaliśmy kawał dobrej roboty – najważniejsze jest już za nami. Nie oznacza to jednak, że nasz projekt jest w pełni gotowy. Do zrobienia pozostało nam jeszcze kilka szczegółów. Nie zajęliśmy się do tej pory paskiem tytułu. Obecnie pokazuje on standardową nazwę nowo tworzonego okna – **Form1** – i standardową



ikonę, jaką Visual Studio nadaje aplikacjom. Powinniśmy spersonalizować naszą grę. Okno gry – podobnie jak wszystkie elementy, które przeciągnęliśmy na okno z przybor-



nika – ma swoje właściwości. Wśród nich możemy znaleźć właściwości **Icon** i **Text**. Druga z nich to napis na pasku tytułu – możemy zmienić jej wartość, wpisując nazwę naszej gry. Pierwsza zaś pozwala na otwarcie okna dialogowego. Z jego poziomu mamy możliwość wybrania pliku – musi to być



jednak plik z rozszerzeniem **.ico**. Ikony to dość specyficzne grafiki o małych wymiarach – dlatego do ich tworzenia dobrze jest wykorzystać specjalny program służący właśnie do

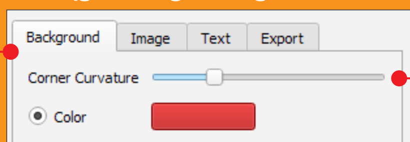
TWORZENIE IKONY

Do tworzenia ikon na własne potrzeby dobrze sprawdzi się także program **3StepIcon Lite** (**DVD-KOD: 001**) (aplikacji tej można używać również do tworzenia ikon na urządzenia mobilne). Nasza ikona będzie miała kształt kwadratu z zaokrąglonymi rogami, a na niej umieścimy na przykład pierwszą literę nazwy gry.



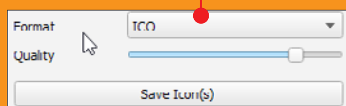
1 Kolor figury wybieramy w zakładce **Background**.

2 Za pomocą suwaka o nazwie **Corner Curvature** możemy regulować zaokrąglenie rogów w figurze.



3 Po przejściu do zakładki **Text** możemy dodać napis na ikonę, określić rodzaj czcionki, a także jej wielkość.

4 W ostatniej zakładce wybieramy format ikony **ICO** i możemy zapisać obraz.

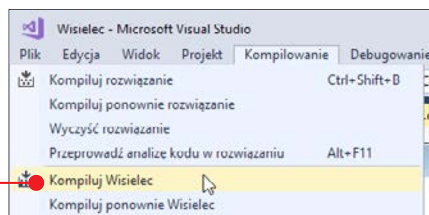


tworzenia ikon, na przykład **IcoFX Portable** (DVD-KOD: 010), **Greenfish Icon Editor Pro** (DVD-KOD: 009) lub **3StepIcon Lite** (DVD-KOD: 001).

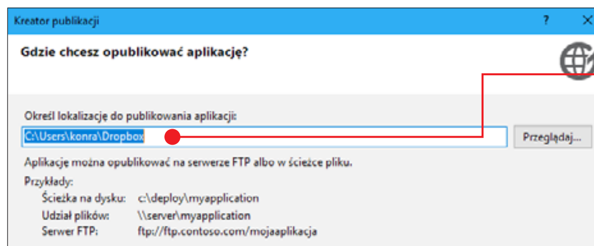
Dzielenie się projektem

Kiedy gra jest gotowa, na pewno chętnie podzielimy się nią ze znajomymi.

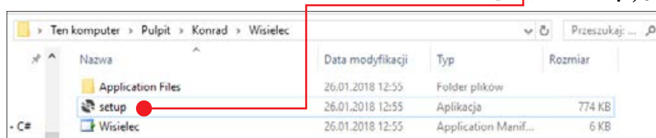
1 By uruchomić kreator tworzenia instalatora gry, w menu głównym Visual Studio



klikamy na **Kompilowanie** i wybieramy **Kompiluj Wisielec**.



2 W nowym oknie określamy lokalizację, gdzie ma znaleźć się gotowy projekt, i przechodzimy przez kolejne kroki kreatora. Kiedy proces publikacji zostanie zakończony, we wskazanym przez nas folderze znajdziemy naszą grę. Klikając na plik **setup**, możemy ją zainstalować.

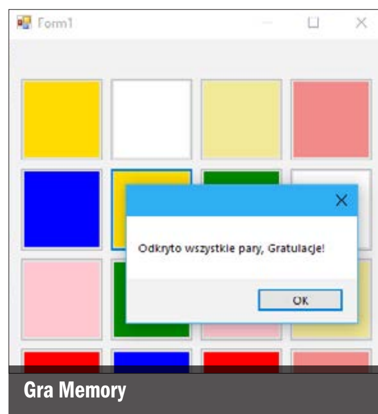


Gra 2. Memory

Kolejną grą, którą stworzymy, poznając język C# i narzędzie Visual Studio, będzie nasza wersja popularnego Memo (memo to skrót od słowa memory, co po angielsku oznacza pamięć – w grze chodzi o zapamiętywanie). Gra ta istniała jeszcze przed erą komputerów, a do rozgrywki wystarczał zestaw par obrazków, które układano losowo grafiką do dołu. Gracz w jednym ruchu odkrywa dwa obrazki. Jeśli są takie same, pozostają odkryte, jeśli różne – trzeba je z powrotem zakryć. Zadaniem gracza jest odkrycie wszystkich pól w jak najmniejszej liczbie ruchów (przydaje się zapamiętywanie, co jest na odkrytych na chwilę obrazkach).

Jak będzie wyglądała nasza gra

Okno gry będzie składało się wyłącznie z przycisków. W przykładowej grze będzie ich



Gra Memory

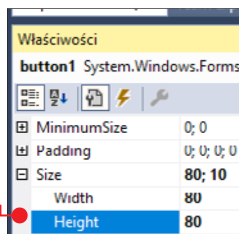
osiem par. Oznacza to, że w oknie powinno znaleźć się 16 przycisków. Nie będziemy wykorzystywać żadnych innych elementów z przybornika. W naszej grze pod przyciskami zamiast obrazków będą kolorowe pola.

„Odkrycie karty” będzie dla nas oznaczało zmianę koloru przycisku. W każdej rozgrywce przyciski będą odkrywały inne kolory – układ kolorów będzie pseudolosowy, dzięki czemu unikniemy powtarzalności. Każda rozgrywka będzie unikalna, ponieważ kolory dla przycisków będą losowane w momencie uruchomienia gry.

Modelujemy okno gry

Po utworzeniu nowego projektu i nadaniu mu nazwy możemy przystąpić do modelowania okna gry. Przeciągamy z przybornika do okna jeden przycisk (kontrolka **Button**). Odpowiednio go sformatujemy (zmienimy jego właściwości), a potem skopiujemy odpowiednią liczbę razy.

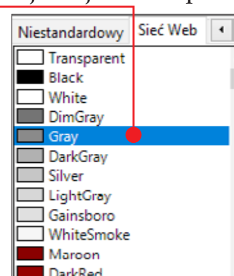
1 Zmieniamy właściwość **Size**, która składa się z dwóch wartości – **Width** (szerokość) i **Height** (wysokość). Aby przycisk był kwadratowy, w obydwu polach powinniśmy umieścić taką samą liczbę, w naszym przykładzie – **80 pikseli**.



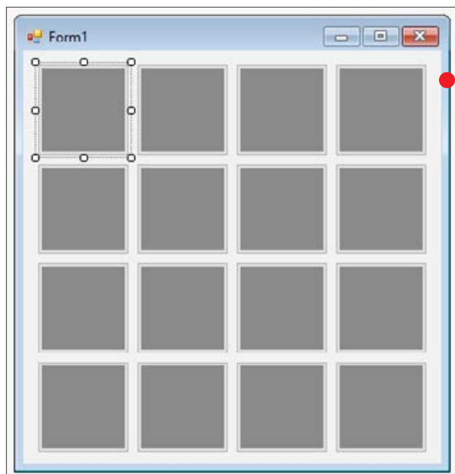
2 Nasze przyciski nie powinny mieć żadnych napisów. Dlatego kasujemy napis z właściwości **Text** i pozostawiamy tam puste pole.



3 Możemy też w przycisku zmienić właściwość **BackColor**. Pozwala ona na ustawienie koloru z jednej z trzech palet: **Niestandardowy**, **Sieć Web**, **System**, które wybieramy z zakładek. Będzie to nasz kolor bazowy. Kiedy wszystkie karty będą zakryte – przyciski będą miały właśnie taką barwę. Kliknięcie na przycisk zmieni wartość we właściwości **BackColor** – co będzie oznaczało odkrycie karty.



4 Tak sformatowany przycisk zaznaczamy, kopiujemy (ctrl+C) i wklejamy (ctrl+V) 15 razy do modelu okna. Następnie rozmieszczamy przyciski w oknie. Możemy ułożyć je w czterech kolumnach po cztery wiersze.



5 Jeśli wszystko poszło dobrze, powinniśmy mieć 16 przycisków o nazwach od **button1** do **button16**. Nie ma dla nas znaczenia to, gdzie umieścimy przycisk o danej nazwie. Ważne jednak, aby kontrolować nazwy przycisków, które są w naszym projekcie – będzie to istotne na kolejnych etapach tworzenia gry.

Nadajemy kolory przyciskom

Kiedy okno gry zostało już przez nas zamodelowane, możemy przejść do pisania kodu. Wystarczy dwukrotne kliknięcie na jeden z naszych przycisków, a Visual Studio przeniesie nas wprost do kodu, w którym wygeneruje też metodę, która wywoływana będzie w momencie kliknięcia na przycisk przez gracza. My na razie pozostawimy tę metodę pustą. Dlaczego? Ponieważ po kliknięciu przycisk powinien zmienić kolor, czyli swoją właściwość **BackColor** – a my jeszcze nie wiemy,

jaką barwę przybierze. Po uruchomieniu naszej gry powinniśmy ustalić, który przycisk w jaki sposób będzie zmieniał kolor. Ustalimy pewien zestaw kolorów, a następnie będziemy wykonywać losowanie przycisków, do których dany kolor zostanie przypisany. W tym celu zdefiniujemy nową metodę, która wylosuje przycisk dla podanego koloru.

1 Definiujemy metodę ***losuj_przycisk()*** – będzie ona losowała numer przycisku (przyciski są ponumerowane tak, jak ich nazwy – od 1 do 16) dla podanego koloru. W parametrze metody będziemy przekazywać kolor, jaki zostanie przypisany do wylosowanego przycisku.

```
private void losuj_przycisk(Color kolor)
{

}
```

2 Wewnątrz metody definiujemy generator, który pozwoli nam na wylosowanie liczby. Za pomocą metody ***Next()*** naszego generatora dostajemy liczbę, która jest losowana z zakresu od 1 w górę i jest mniejsza niż 17. Wynik losowania przypisany jest do nowo utworzonej zmiennej, nazwanej przez nas **numer_pola**.

3 Teraz należałoby przypisać podany w parametrze kolor do przycisku o wylosowanym numerze. Nie możemy jednak jeszcze tego zrobić. Stworzona przez nas metoda będzie wywoływana wielokrotnie – do wylosowania przycisku dla każdego z podanych kolorów. Musimy uwzględnić to w jej treści – tak aby była ona uniwersalna.
W wypadku gdy losujemy numer kolejny raz, może nastąpić sytuacja, że wylosowany numer ma już przypisany kolor. A w takim razie powinniśmy ponowić losowanie.
Niezbędne do tego okaże się utworzenie tablicy, w której polom o indeksach odpowiadających numerom przycisków przypiszemy informacje o tym, czy dla danego przycisku kolor został już przypisany. W związku z tym przenosimy się do góry naszej klasy – tablica, którą utworzymy, powinna być zadeklarowana jako pole klasy. Tablicę nazwiemy **czy_przyciski_maja_kolor** i nadamy jej typ **bool** (to oznacza, że wartości, jakie mogą przyjmować jej elementy, to **true** – prawda i **false** – fałsz). Tablica będzie miała 17 elementów – poindeksowanych od **0** do **16**. Pierwszy element, z indeksem **0**, nie będzie przez nas wykorzystywany, ale dzięki temu możemy użyć pozostałych 16 elementów, których indeksy będą odpowiadać numerom przycisków.

```
namespace Memory
{
    public partial class Form1 : Form
    {
        bool[] czy_przyciski_maja_kolor = new bool[17];
```

4 Wracamy do metody ***losuj_przycisk()*** – losowanie numeru przycisku powinno być zapętlone i wykonywane, dopóki w tablicy **czy_przyciski_maja_kolor** pod indeksem określonym wylosowaną wartością będzie znajdować się wartość **true**. Możemy do tego wykorzystać znaną już nam pętlę **while**.

```
private void losuj_przycisk(Color kolor)
{
    Random gen = new Random();
    int numer_pola = gen.Next(1, 17);
    while (czy_przyciski_maja_kolor[numer_pola] == true)
    {
        numer_pola = gen.Next(1, 17);
    }
```

Będzie ona ponawiać wykonywanie losowania numeru pola aż do trafienia na pole oznaczające przycisk, dla którego kolor nie został jeszcze ustawiony.

5 Po zakończeniu wykonywania się pętli dla numeru, który został wylosowany i zapisany jako wartość zmiennej **numer_pola**, w tablicy mówiącej o tym, które przyciski mają już wyznaczone kolory, wstawiamy wartość **true**.

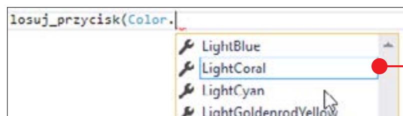
```
private void losuj_przycisk(Color kolor)
{
    Random gen = new Random();
    int numer_pola = gen.Next(1, 17);
    while (czy_przyciski_maja_kolor[numer_pola] == true)
    {
        numer_pola = gen.Next(1, 17);
    }
    czy_przyciski_maja_kolor[numer_pola] = true;
```

```
namespace Memory
{
    public partial class Form1 : Form
    {
        bool[] czy_przyciski_maja_kolor = new bool[17];
        Color[] kolory = new Color[17];
```

6 Oprócz tego, czy przycisk ma już wylosowany kolor, drugim ważnym zagadnieniem jest przechowanie informacji o tym, jaki kolor powinien być przypisany do danego przycisku. Nie możemy do tego wykorzystać posiadanej tablicy (przechowuje ona wartości boolowskie). Musimy mieć nową tablicę, w której przechowywane będą kolory. Tablica ta powinna być polem naszej klasy. Dlatego przenosimy się na górę skryptu i deklarujemy tam tablicę **kolory**, której typem będzie **Color**. Tak naprawdę, to jest to tablica obiektów klasy **Color**. Obiektów tych używamy już w parametrze metody ***losuj_przycisk***. I właśnie kolor przekazany przez parametr podczas wywołania metody powinien trafić do tablicy kolorów pod indeksem o wytypowanym numerze pola.

```
private void losuj_przycisk(Color kolor)
{
    Random gen = new Random();
    int numer_pola = gen.Next(1, 17);
    while (czy_przyciski_maja_kolor[numer_pola] == true)
    {
        numer_pola = gen.Next(1, 17);
    }
    czy_przyciski_maja_kolor[numer_pola] = true;
    kolory[numer_pola] = kolor;
}
```

7 Przyciski powinny mieć przyporządkowane kolory już na starcie gry. Oznacza to, że stworzoną przez nas metodę powinniśmy wywołać w konstruktorze. I to nie raz, ale szesnaście razy, ponieważ każdy przycisk musi otrzymać jakiś kolor, a jedno wywołanie wybiera kolor tylko dla jednego przycisku. Pamiętajmy o tym, że nasza gra polega na odnajdywaniu par – dlatego każdy kolor powinien pojawić się na dwóch przyciskach. Nazwy kolorów podajemy w języku angielskim, wpisując najpierw słowo **Color**,



a potem stawiając znak kropki. Podczas pisania Visual Studio wyświetla podpowiedzi, z jakich barw możemy skorzystać.

Odkrywamy karty

Stworzony przez nas kod wciąż pozostaje w strefie logiki naszej aplikacji. Uruchomienie programu nie da nam jeszcze żadnego realnego efektu. Co więcej – nie zobaczymy, jaki kolor został przypisany do którego przycisku, ponieważ przyciski nie zmieniły swoich kolorów ani nie reagują na kliknięcia.

```
namespace Memory
{
    public partial class Form1 : Form
    {
        bool[] czy_przyciski_maja_kolor = new bool[17];
        Color[] kolory = new Color[17];
        bool[] zakryte = new bool[17];
```

1 Powinniśmy zatem napisać metodę, która sprawi, że przyciski zmienią swoje kolory. Będzie to uniwersalna metoda, którą wykorzystamy kilka razy. Dlaczego? Po każdym odkryciu dwóch kolorów będziemy musieli pozostawić przyciski w zmienionych kolorach (gdy odnaleziono parę) lub przywrócić im szary kolor (gdy pary nie odnaleziono). Powinniśmy zatem przechowywać informacje o tym, które przyciski

```
public Form1()
{
    InitializeComponent();
    losuj_przycisk(Color.Red);
    losuj_przycisk(Color.Red);
    losuj_przycisk(Color.Blue);
    losuj_przycisk(Color.Blue);
    losuj_przycisk(Color.Pink);
    losuj_przycisk(Color.Pink);
    losuj_przycisk(Color.Green);
    losuj_przycisk(Color.Green);
    losuj_przycisk(Color.LightCoral);
    losuj_przycisk(Color.LightCoral);
    losuj_przycisk(Color.Gold);
    losuj_przycisk(Color.Gold);
    losuj_przycisk(Color.Khaki);
    losuj_przycisk(Color.Khaki);
    losuj_przycisk(Color.White);
    losuj_przycisk(Color.White);
}
```

```
private void koloruj_przyciski()
{
    if (zakryte[1] == false) { button1.BackColor = kolory[1]; } else { button1.BackColor = Color.Gray; }
    if (zakryte[2] == false) { button2.BackColor = kolory[2]; } else { button2.BackColor = Color.Gray; }
    if (zakryte[3] == false) { button3.BackColor = kolory[3]; } else { button3.BackColor = Color.Gray; }
    if (zakryte[4] == false) { button4.BackColor = kolory[4]; } else { button4.BackColor = Color.Gray; }
    if (zakryte[5] == false) { button5.BackColor = kolory[5];} else { button5.BackColor = Color.Gray; }
    if (zakryte[6] == false) { button6.BackColor = kolory[6];} else { button6.BackColor = Color.Gray; }
    if (zakryte[7] == false) { button7.BackColor = kolory[7];} else { button7.BackColor = Color.Gray; }
    if (zakryte[8] == false) { button8.BackColor = kolory[8];} else { button8.BackColor = Color.Gray; }
    if (zakryte[9] == false) { button9.BackColor = kolory[9];} else { button9.BackColor = Color.Gray; }
    if (zakryte[10] == false) { button10.BackColor = kolory[10];} else { button10.BackColor = Color.Gray; }
    if (zakryte[11] == false) { button11.BackColor = kolory[11];} else { button11.BackColor = Color.Gray; }
    if (zakryte[12] == false) { button12.BackColor = kolory[12];} else { button12.BackColor = Color.Gray; }
    if (zakryte[13] == false) { button13.BackColor = kolory[13];} else { button13.BackColor = Color.Gray; }
    if (zakryte[14] == false) { button14.BackColor = kolory[14];} else { button14.BackColor = Color.Gray; }
    if (zakryte[15] == false) { button15.BackColor = kolory[15];} else { button15.BackColor = Color.Gray; }
    if (zakryte[16] == false) { button16.BackColor = kolory[16];} else { button16.BackColor = Color.Gray; }
}
```

powinny być zakryte, a które powinny pozostać odkryte. Informacja ta powinna być przechowywana w tablicy. Tworzymy zatem tablicę typu **bool**, którą deklarujemy jako pole klasy (czyli na początku naszego kodu) i nazywamy ją **zakryte**. Na miejscach określonych indeksami odpowiadającymi numerom przycisków będzie znajdować się wartość **true**, gdy przycisk ma być zakryty. Gdy w tablicy pojawi się wartość **false** – przycisk będzie odkryty, czyli dostanie kolor, który został mu przyporządkowany podczas losowań.

```
private void koloruj_przyciski()
{ }
```

2 Tworzymy zatem metodę ***koloruj_przyciski()***. W jej treści będą znajdować się instrukcje warunkowe sprawdzające, czy w tablicy **zakryte** znajduje się wartość **false**. Jeśli tak, zmieniamy właściwość **BackColor** przycisku o numerze odpowiadającym indeksowi sprawdzanego przez nas pola z tablicy na kolor znajdujący się w tablicy z kolorami pod tym samym indeksem. Jeśli warunek ten nie zostanie spełniony – ten sam przycisk powinien dostać kolor startowy, czyli szary, który ustawiliśmy, modelując okno gry.

3 Stworzona przez nas metoda będzie wywoływana po kliknięciu na przycisk i sprawdzeniu, czy należy zmieniać wartość w tablicy **zakryte**. Gdybyśmy teraz chcieli skorzystać z tej metody, może to być trudne, ponieważ tablica **zakryte** nie jest wypełniona żadnymi wartościami. Powinniśmy ją wypełnić wartościami **true**. Możemy zrobić to w pętli, dopisanej do konstruktora. Skorzystamy przy tym ze znanej nam już pętli **for** – która iterować będzie na zmiennej **i**

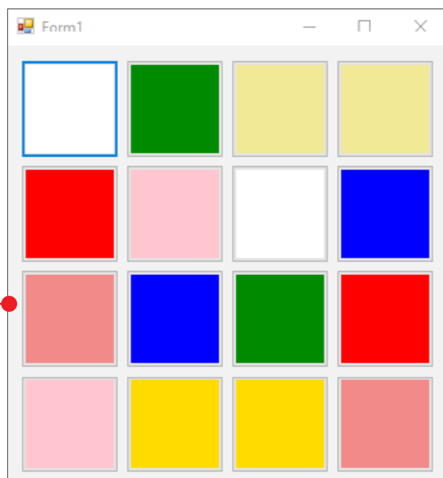
```
public Form1()
{
    InitializeComponent();
    losuj_przycisk(Color.Red);
    losuj_przycisk(Color.Red);
    losuj_przycisk(Color.Blue);
    losuj_przycisk(Color.Blue);
    losuj_przycisk(Color.Pink);
    losuj_przycisk(Color.Pink);
    losuj_przycisk(Color.Green);
    losuj_przycisk(Color.Green);
    losuj_przycisk(Color.LightCoral);
    losuj_przycisk(Color.LightCoral);
    losuj_przycisk(Color.Gold);
    losuj_przycisk(Color.Gold);
    losuj_przycisk(Color.Khaki);
    losuj_przycisk(Color.Khaki);
    losuj_przycisk(Color.White);
    losuj_przycisk(Color.White);

    for (int i = 1; i < 17; i++)
    {
        zakryte[i] = true;
    }
}
```

– wykorzystamy ją do opisania indeksów, dzięki czemu łatwo odniesiemy się do każdego elementu tablicy **zakryte**.

4 Gdybyśmy wypełnili naszą tablicę **zakryte** wartościami **false**, a następnie uruchomili napisaną przez nas metodę ***koloruj_przyciski()*** – po uruchomieniu

```
    for (int i = 1; i < 17; i++)
    {
        zakryte[i] = false;
    }
    koloruj_przyciski();
}
```



naszej gry mielibyśmy teraz podgląd na to, jakie kolory zostały przypisane do których przycisków – wszystkie przyciski zostałyby „pokolorowane”.

5 Powinniśmy jednak przywrócić konstruktor do wcześniejszego stanu, a zmianę kolorów przycisków przenieść do metod wywoływanych po kliknięciu na każdy z przycisków. Przenosimy się zatem do

```
public Form1()
{
    InitializeComponent();
    losuj_przycisk(Color.Red);
    losuj_przycisk(Color.Red);
    losuj_przycisk(Color.Blue);
    losuj_przycisk(Color.Blue);
    losuj_przycisk(Color.Pink);
    losuj_przycisk(Color.Pink);
    losuj_przycisk(Color.Green);
    losuj_przycisk(Color.Green);
    losuj_przycisk(Color.LightCoral);
    losuj_przycisk(Color.LightCoral);
    losuj_przycisk(Color.Gold);
    losuj_przycisk(Color.Gold);
    losuj_przycisk(Color.Khaki);
    losuj_przycisk(Color.Khaki);
    losuj_przycisk(Color.White);
    losuj_przycisk(Color.White);

    for (int i = 1; i < 17; i++)
    {
        zakryte[i] = true;
    }
}
```

```
private void button1_Click(object sender, EventArgs e)
{

}
```

procedury wygenerowanej poprzez kliknięcie na przycisk podczas modelowania gry.

6 Po kliknięciu na przycisk powinniśmy zmienić wartość jego właściwości **BackColor** na kolor przechowywany pod indek-

```
private void button1_Click(object sender, EventArgs e)
{
    button1.BackColor = kolory[1];
}
```

sem odpowiadającym numerowi przycisku w tablicy z kolorami.

7 Dalsze zachowanie naszego programu jest zależne od tego, czy odkrywamy kartę pierwszą z pary, czy drugą. Jeśli jako wykonanie ruchu postrzegamy odkrycie dwóch kart, to po odkryciu pierwszej powinna ona pozostać odkryta aż do czasu odkrycia drugiej karty, po czym powinniśmy wykonać sprawdzenie, czy odnaleźliśmy parę. Następnie zależnie od wyniku sprawdzenia należy zakryć lub pozostawić odkryte obydwie karty. Trzeba zatem w jakiś sposób przechować informację, czy odkrywamy pierwszą, czy drugą kartę w parze. Stworzymy w tym celu zmienną (pole **klasy**). Nazwiemy je **czy_pierwszy_klik** – będzie ono mogło przyjąć wartość **true** lub **false** – wybranym przez nas typem będzie zatem

```
bool[] czy_przyciski_maja_kolor = new bool[17];
Color[] kolory = new Color[17];
bool[] zakryte = new bool[17];
bool czy_pierwszy_klik = true;
```

bool. Do pola przypisujemy od razu wartość **true**, bo taka będzie wartość przy uruchomieniu gry – ponieważ na starcie zaczynamy od pierwszego kliknięcia, a nie od drugiego.

8 Po kliknięciu na przyciski w oknie programu sprawdzamy wartość zmiennej **czy_pierwszy_klik**. Jeśli jest ona równa **true**, będziemy wykonywać inne instrukcje,

```
private void button1_Click(object sender, EventArgs e)
{
    button1.BackColor = kolory[1];
    if (czy_pierwszy_klik == true)
    {

    }
    else
    {
    }
}
```

niż kiedy będzie ona równa **false**. Dlatego budujemy instrukcję warunkową, w której znajduje się też sekcja **else**.

9 Najpierw zajmiemy się tym, co dzieje się, kiedy gracz wykona ruch, czyli kliknie na dwa przyciski. Powinniśmy zachować informację o numerach przycisków, na które kliknął – pierwszego i drugiego. Będzie to potrzebne do sprawdzenia, czy trafił parę. Zatem dodajemy w naszej klasie dwa pola będące liczbami całkowitymi (typ **int**). Jedno będzie przechowywać numer pierwszego przycisku, na który kliknięto w danym ruchu, a drugie – numer drugiego przycisku.

```
namespace Memory
{
    public partial class Form1 : Form
    {
        bool[] czy_przyciski_maja_kolor = new bool[17];
        Color[] kolory = new Color[17];
        bool[] zakryte = new bool[17];
        bool czy_pierwszy_klik = true;
        int ruchy = 0;
        int numer_pierwszego = 0;
        int numer_drugiego = 0;
```

10 Numer przycisku przypisujemy do zmiennej **numer_pierwszego**, a następnie zmieniamy wartość zmiennej **czy_pierwszy_klik** na **false**.

```
private void button1_Click(object sender, EventArgs e)
{
    button1.BackColor = kolory[1];
    if (czy_pierwszy_klik == true)
    {
        numer_pierwszego = 1;
        czy_pierwszy_klik = false;
    }
    else
    {
    }
}
```

11 Kliknięcie na kolejny przycisk będzie już drugim kliknięciem. Zachowanie przycisku podczas drugiego kliknięcia opisujemy w sekcji **else**. Zaczynamy od zapisania numeru przycisku do zmiennej **numer_drugiego** i przypisania zmiennej **czy_pierwszy_klik** ponownie wartości **true**

```
private void button1_Click(object sender, EventArgs e)
{
    button1.BackColor = kolory[1];
    if (czy_pierwszy_klik == true)
    {
        numer_pierwszego = 1;
        czy_pierwszy_klik = false;
    }
    else
    {
        numer_drugiego = 1;
        czy_pierwszy_klik = true;
    }
}
```

– ponieważ po wykonaniu drugiego kliknięcia w jednym ruchu kolejne kliknięcie będzie już pierwszym kliknięciem następnego ruchu.

Sprawdzenie pary

1 To jeszcze nie koniec instrukcji, jakie trzeba wykonać po drugim kliknięciu w danym ruchu. Należy uruchomić sprawdzanie, czy trafiono w parę, a także pokolorować przyciski. Sprawdzanie powinno określić, czy mają być zmienione wartości w tablicy **zakryte**, a ***koloruj_przyciski()*** powinno te zmiany uwzględnić. Do sprawdzania wykorzystamy metodę ***sprawdz()***. **Po jej wpisaniu linia ta zostanie podkreślona (co oznacza błąd w kodzie) – ponieważ metoda ta jeszcze nie istnieje.** Jej napisa-

```
button1.BackColor = kolory[1];
if (czy_pierwszy_klik == true)
{
    numer_pierwszego = 1;
    czy_pierwszy_klik = false;
}
else
{
    numer_drugiego = 1;
    czy_pierwszy_klik = true;
    sprawdz();
    koloruj_przyciski();
}
```

```
private void button1_Click(object sender, EventArgs e)
{
    if (button1.BackColor != kolory[1])
    {
        button1.BackColor = kolory[1];
        if (czy_pierwszy_klik == true)
        {
            numer_pierwszego = 1;
            czy_pierwszy_klik = false;
        }
        else
        {
            numer_drugiego = 1;
            czy_pierwszy_klik = true;
            sprawdz();
            koloruj_przyciski();
        }
    }
}
```

niem zajmiemy się po zakończeniu opisywania kliknięć na przyciski.

2 Powinniśmy jeszcze nieco zmodyfikować metodę wywoływaną po kliknięciu na przycisk. Teraz, gdyby gracz zechciał w jednym ruchu kliknąć dwa razy na to samo pole, gra by mu na to pozwoliła. Musimy temu zapobiec – jeden ruch to dwa kliknięcia na różne pola, a nie na to samo. Na początku naszej metody powinniśmy zatem sprawdzić (poprzez instrukcję **if**), czy właściwość **BackColor** w klikanym przycisku ma inną wartość niż kolor kryjący się w tablicy z kolorami pod indeksem takim jak numer klikanego przycisku. Jeśli tak, możemy wykonać cały wcześniej wprowadzony kod.

3 Stworzony przez nas kod powinien być z drobnymi modyfikacjami wykonywany po kliknięciu na każdy z 16 przycisków. Należy wygenerować metody odpowiedzialne za kliknięcie na każdy przycisk, a w ich treści uwzględnić przycisk, do którego się odnosimy, a także jego numer i indeks w tablicy kolorów.

4 Teraz możemy przystąpić do definicji metody ***sprawdz()***. Jej zadaniem będzie nie tylko sprawdzenie, czy trafiliśmy na parę

```
private void sprawdz()
{ }
```

w jednym ruchu, ale także sprawdzenie, czy odkryto już wszystkie karty.

Object Browser | Form1.cs* | Form1.cs [Design]

Memory | Memory.Form1

```
                numer_drugiego = 2;
                czy_pierwszy_klik = true;
                sprawdz();
                koloruj_przyciski();
            }
        }
    }

    private void button3_Click(object sender, EventArgs e)
    {
        if (button3.BackColor != kolory[3])
        {
            button3.BackColor = kolory[3];
            if (czy_pierwszy_klik == true)
            {
                numer_pierwszego = 3;
                czy_pierwszy_klik = false;
            }
            else
            {
                numer_drugiego = 3;
                czy_pierwszy_klik = true;
                sprawdz();
                koloruj_przyciski();
            }
        }
    }

    private void button4_Click(object sender, EventArgs e)
    {
        if (button4.BackColor != kolory[4])
        {
            button4.BackColor = kolory[4];
            if (czy_pierwszy_klik == true)
            {
                numer_pierwszego = 4;
                czy_pierwszy_klik = false;
            }
            else
            {
                numer_drugiego = 4;
                czy_pierwszy_klik = true;
                sprawdz();
```

5 Nie możemy zacząć od sprawdzenia, czy odkryto już wszystkie karty, to będzie możliwe dopiero, gdy zaprogramujemy sprawdzanie trafienia w pojedynczą parę. Takie sprawdzanie polega na podejrzeniu tego, czy kolory przycisków, które odkryliśmy w danym ruchu, są takie same. Pomocna okaże nam się tablica z kolorami przycisków, a także przechowywane pod **numer_pierwszego** i **numer_drugiego** numery przycisków, których użyjemy jako indeksów, pod którymi przyrównamy kolory z tablicy. Do przyrównania wykorzystujemy instrukcję warunkową.

```
private void sprawdz()
{
    if (kolory[numer_pierwszego] == kolory[numer_drugiego])
    {

    }
```

```
private void sprawdz()
{
    if (kolory[numer_pierwszego] == kolory[numer_drugiego])
    {
        MessageBox.Show("Brawo! Para odgadnięta!");
    }
```

6 Jeśli napisany przez nas warunek jest spełniony, poprzez polecenie ***MessageBox.Show()*** wyświetlamy graczowi komunikat z informacją o tym, że para została odnaleziona.

7 Odnaleziona para powinna pozostać odkryta – to znaczy, że w tablicy przechowującej informacje o tym, czy dane pola mają być zakryte, pod indeksami o numerach przycisków, gdzie odnaleziono parę, wstawiamy wartości **false**.

```
private void sprawdz()
{
    if (kolory[numer_pierwszego] == kolory[numer_drugiego])
    {
        MessageBox.Show("Brawo! Para odgadnięta!");
        zakryte[numer_pierwszego] = false;
        zakryte[numer_drugiego] = false;
    }
```

8 Naszą instrukcję warunkową kończymy, dodając do niej sekcję **else**, w której ponownie wykorzystujemy polecenie ***MessageBox.Show()*** – jednak tym razem informujemy gracza o tym, że para nie została odnaleziona.

```
private void sprawdz()
{
    if (kolory[numer_pierwszego] == kolory[numer_drugiego])
    {
        MessageBox.Show("Brawo! Para odgadnięta!");
        zakryte[numer_pierwszego] = false;
        zakryte[numer_drugiego] = false;
    }
    else
    {
        MessageBox.Show("Niestety, próbuj dalej");
    }
}
```

9 To jeszcze nie koniec definicji metody ***sprawdz()***. Drugim zadaniem, które powinna ona wykonywać, jest sprawdzenie, czy wszystkie pary zostały już odkryte. Będziemy w niej zliczać, ile pól zostało odkrytych, a wartość tę przechowamy w zmiennej **odkryte**, którą tworzymy wewnątrz metody (dajemy jej typ **int**, ponieważ to liczba całkowita). Następnie tworzymy pętlę **for**, która pozwoli nam na przejście po wszystkich elementach tablicy **zakryte** – jeśli pod którymś z nich będzie kryła się wartość **false**, to zwiększymy wartość zmiennej **odkryte** o jeden (**odkryte++**).

```
    else
    {
        MessageBox.Show("Niestety, próbuj dalej");
    }

    int odkryte = 0;
    for (int g = 1; g < 17; g++)
    {
        if (zakryte[g] == false)
        {
            odkryte++;
        }
    }
```

10 Kiedy zmienna **odkryte** osiągnie wartość **16** – będzie to oznaczać, że każde z pól zostało już odkryte, a gra zosta-

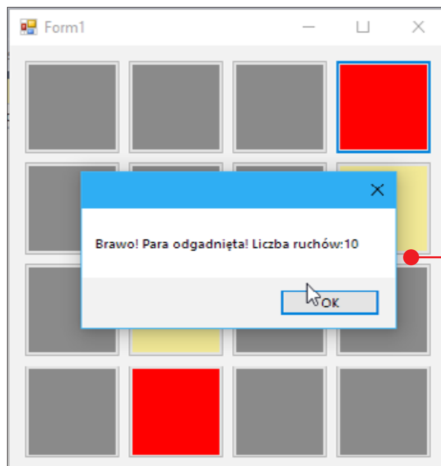
ła zakończona. Za pomocą instrukcji warunkowej **if** przyrównujemy wartość zmiennej **odkryte** do szesnastu i jeśli nasze porównanie jest prawdą, ponownie poprzez polecenie ***MessageBox.Show()*** wyświetlimy graczowi komunikat z gratulacjami, mówiący o tym, że odkrył już wszystkie karty.

```
private void sprawdz()
{
    if (kolory[numer_pierwszego] == kolory[numer_drugiego])
    {
        MessageBox.Show("Brawo! Para odgadnięta!");
        zakryte[numer_pierwszego] = false;
        zakryte[numer_drugiego] = false;
    }
    else
    {
        MessageBox.Show("Niestety, próbuj dalej");
    }

    int odkryte = 0;
    for (int g = 1; g < 17; g++)
    {
        if (zakryte[g] == false)
        {
            odkryte++;
        }
    }
    if (odkryte == 16) { MessageBox.Show("Odkryto wszystkie pary, Gratulacje!"); }
}
```

11 Aby dopełnić naszą metodę, możemy dodać zliczanie ruchów wykonanych przez gracza, potrzebnych mu do odgadnięcia wszystkich par. Powinniśmy zliczać wykonane przez gracza ruchy. Aby przechowywać ich liczbę, deklarujemy pole **ruchy** (typ **int**) w klasie. Wartość początkowa tego pola to **0**.

```
namespace Memory
{
    public partial class Form1 : Form
    {
        bool[] czy_przyciski_maja_kolor = new bool[17];
        Color[] kolory = new Color[17];
        bool[] zakryte = new bool[17];
        bool czy_pierwszy_klik = true;
        int ruchy = 0;
        int numer_pierwszego = 0;
        int numer_drugiego = 0;
```



12 Każde wywołanie sprawdzenia oznacza wykonanie kolejnego ruchu, dlatego na początku metody ***sprawdz()*** dopisujemy zwiększenie wartości pola **ruchy**. Wartość tego pola powinna być do wypisywania. Możemy to robić z każdym odnalezieniem przez gracza pary – modyfikując nieco tekst, który wyświetlamy poprzez **MessageBox**.

Nasza gra już działa. Możemy teraz zmienić ikonę i napis na pasku tytułowym, analogicznie jak w opisie gry Wisielec, i przejść do testowania.

```
private void sprawdz()
{
    ruchy = ruchy + 1;
    if (kolory[numer_pierwszego] == kolory[numer_drugiego])
    {
        MessageBox.Show("Brawo! Para odgadnięta! Liczba ruchów: " + ruchy);
        zakryte[numer_pierwszego] = false;
        zakryte[numer_drugiego] = false;
```

Gra 3. Snake

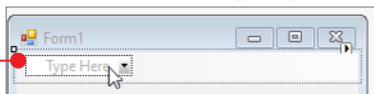
Kolejną grą, jaką napiszemy w języku C#, będzie klon kultowego Snake'a – gry znanej z dawnych modeli telefonów komórkowych oraz z licznych serwisów internetowych z grami online.
Zasady gry są bardzo proste. Polega ona na sterowaniu wężem, który może poruszać się w czterech kierunkach. Musimy tak go prowadzić, aby nie uderzył sam w siebie i po drodze zbierał owoce. Zebranie owocu wydłuża węża. Celem gry jest takie kierowanie wężem, by stał się jak najdłuższy.
Nasz Snake będzie – podobnie jak wcześniej stworzone przez nas gry – aplikacją desktopową. Dlatego tworzymy nowy projekt, podobnie jak w przypadku gier Memory oraz Wisielec – jako **Aplikacja Windows Forms**.

Modelowanie okna gry

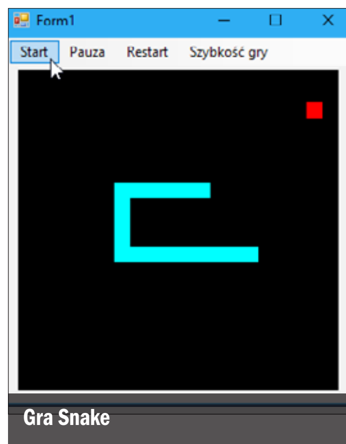
Tym razem podczas modelowania okna gry skorzystamy jedynie z dwóch elementów znalezionych w Przyborniku (nie oznacza to jednak, że będą to jedyne rzeczy, jakie weźmiemy z Przybornika).

1 Okno naszej gry będzie składać się z elementów **Panel** i **MenuStrip** – dlatego przeciągamy oba z Przybornika na model okna.

2 **MenuStrip** automatycznie ustawia się u góry okna i pozwala w polach **Type Here** wpisywać teksty, jakie mają pojawić się na przyciskach menu. Wpisanie jednego tekstu na przycisk powoduje pojawienie się

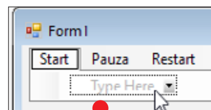


nowego pola **Type Here**, w które możemy wpisać kolejny tekst. Przyciski nie muszą pojawiać się jedynie obok siebie. Podobnie jak w wielu aplikacjach, po najechaniu na przycisk menu może rozwinąć się lista opcji

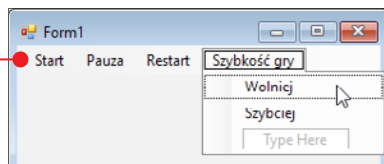


Gra Snake

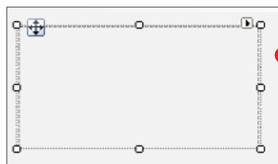
do wyboru – stworzymy ją, wpisując teksty na przyciski w polach **Type Here** widocznych po zaznaczeniu przycisku głównego.



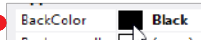
3 Na potrzeby naszej gry stworzymy menu składające się z czterech opcji głównych: **Start**, **Pauza**, **Restart**, **Szybkość gry**. Ostatnia z pozycji powinna rozwijać się po najechaniu na nią i dawać graczowi możliwość wyboru jednej z dwóch możliwości: **Szybciej** i **Wolniej**.



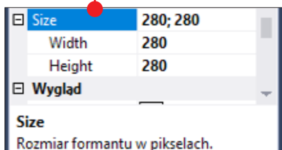
4 Drugi z zaczerpniętych z Przybornika elementów – **Panel** – wykorzystamy jako podstawę, na której wyrysujemy naszą grę. Musimy zmienić dwie właściwości tego elementu.



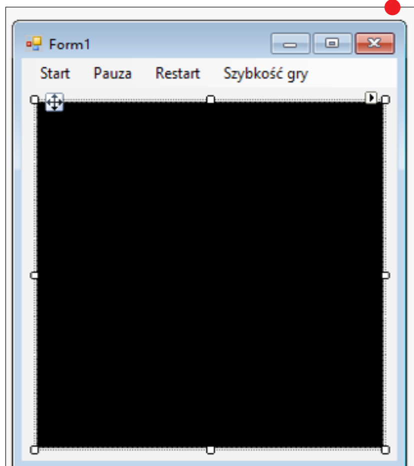
5 W oknie **Właściwości** zmieniamy wartość właściwości **BackColor** na kolor czarny – będzie to kolor tła naszej gry.



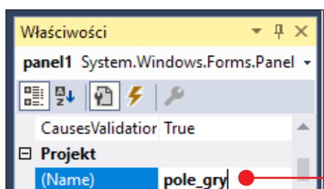
6 Druga właściwość do zmiany to **Size**, czyli rozmiar. Ustawiamy go na wartość 280 na 280 pikseli. Aby cały panel zmieścił



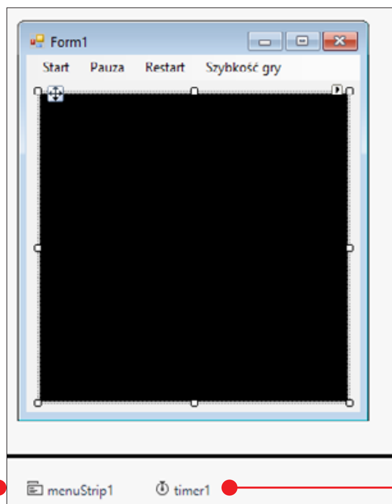
się w modelu okna, zmieniamy nieco rozmiar samego modelowanego okna, aby ostatecznie uzyskać efekt podobny jak na ilustracji.



7 Aby lepiej pokazać, do czego służy Panel, możemy zmienić jego nazwę (właściwość **Name**) na **pole_gry**.



8 Okno jest już zamodelowane, jednak nie skończyliśmy jeszcze dodawać do



projektu elementów z Przybornika. Teraz przeciągamy element **Timer**. Nie jest on widoczny w modelu okna, jednak to, że został dodany, widać pod podglądem. Po co jest nam **Timer**? Daje on nam do dyspozycji zdarzenie **Tick** – jest to zdarzenie wywoływane cyklicznie w określonych odstępach czasu. Wykorzystamy je później do zaprojektowania ruchu naszego węża.

Zanim uruchomimy grę

Gra powinna być włączana przez kliknięcie na przycisk **Start** w menu głównym. Trzeba jednak dać graczowi do zrozumienia, że ma coś zrobić (kliknąć na przycisk). A to znaczy, że nasz projekt musi wyglądać inaczej przed kliknięciem na **Start** niż po kliknięciu. Dodaliśmy już do gry **Timer**, który w określonych odstępach czasu wywołuje zdarzenie – to znaczy wykonuje wskazaną metodę. Domyślnie metoda ta uruchamiana jest co 100 milisekund, czyli całkiem często. Sformułujemy ją tak, aby do kliknięcia na **Start** działała inaczej niż po kliknięciu.

1 Aby przejść do definicji tej metody, klikamy dwukrotnie na **timer1** w podglądzie modelowanego okna gry. Mamy już wygenerowaną metodę przypisaną do zdarzenia **Timera**. Teraz musimy nadać jej odpowiednią treść. Najprostszym rozwiązaniem

```
private void timer1_Tick(object sender, EventArgs e)
{

}
```

```
else
{
    pole_gry.CreateGraphics().DrawString()
}
 ▲ 3 of 6 ▼ void Graphics.DrawString(string s, Font font, Brush brush, float x, float y)
   Draws the specified text string at the specified location with the specified Brush and Font objects.
   s: String to draw.
```

będzie zastosowanie instrukcji warunkowej **if**. Będzie ona sprawdzać, czy gra jest już rozpoczęta, czy też nie. Instrukcja ta w swoim warunku będzie sprawdzać wartość pewnej zmiennej.

2 Tworzymy pole klasy **Form1** typu boolowskiego – ***czy_gra_aktywna***. Po

```
namespace Snake
{
    public partial class Form1 : Form
    {
        private bool czy_gra_aktywna;
```

uruchomieniu gry przypiszemy tam wartość **false**, co będzie oznaczać, że gra nie została rozpoczęta.

3 Przypisania dokonujemy w konstruktorze.

```
public Form1()
{
    InitializeComponent();
    czy_gra_aktywna = false;
}
```

4 Powracamy do metody wywoływanej przez **Timer** i w niej tworzymy instrukcję warunkową, której wykonanie zależne jest od wartości ***czy_gra_aktywna***. Instrukcja ta powinna zawierać także sekcję **else**. Umieszczenie samego ***czy_gra_aktywna*** w warunku jest równoznaczne z napisaniem **czy_gra_aktywna == true**.

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
```

5 W tym momencie interesuje nas czas, gdy gra nie jest rozpoczęta – dlatego dalej będziemy pisać w sekcji **else**. Naszym celem jest wyświetlenie napisu **Naciśnij Start** na panelu **pole_gry**. Aby to zrobić, musimy skorzystać z funkcji **DrawString**. Jedna z form jej wywołania wymaga użycia w parametrze obiektów typu **Font** i **Brush**.

Zanim przystąpimy do wywołania funkcji **DrawString**, musimy mieć te obiekty. Dodatkowo konstruktor obiektu typu **Font** wymaga od nas posiadania obiektu **FontFamily**.

Dlatego pisanie rozpoczynamy od utworzenia obiektu typu **FontFamily** – dajmy mu nazwę **fontFamily1**. W parametrze konstruktora tego obiektu podajemy nazwę czcionki, jaką chcemy stworzyć napis.

```
else
{
    FontFamily fontFamily1 = new FontFamily("Arial");
```

6 Dalej tworzymy obiekt klasy **Font**, który w parametrze konstruktora pobiera stworzony wcześniej obiekt klasy **FontFamily**, a także rozmiar czcionki, jaką chcemy utworzyć napis. Wielkość **15** punktów będzie odpowiednia.

```
FontFamily fontFamily1 = new FontFamily("Arial");
Font f = new Font(fontFamily1, 15);
```

7 Tworząc obiekt typu **Brush**, wykorzystujemy konstruktor klasy **SolidBrush**. Nie możemy skorzystać z konstruktora klasy **Brush**, gdyż klasa ta jest klasą abstrakcyjną. A to dla nas oznacza, że jest klasą bazową dla innych klas po niej dziedziczących (z których jedną jest **SolidBrush**). Konstruktor ten wymaga od nas podania jedynie koloru, jakim chcemy pisać.

```
FontFamily fontFamily1 = new FontFamily("Arial");
Font f = new Font(fontFamily1, 15);
Brush b = new SolidBrush(Color.Aqua);
```

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {

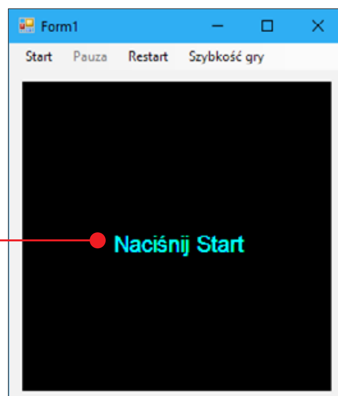
    }
    else
    {
        FontFamily fontFamily1 = new FontFamily("Arial");
        Font f = new Font(fontFamily1, 15);
        Brush b = new SolidBrush(Color.Aqua);
        pole_gry.CreateGraphics().DrawString("Naciśnij Start", f, b, 80, 135);
    }
}
```

A B C D E

8 Dopiero teraz jesteśmy gotowi, aby skorzystać z funkcji **DrawString**, która jest wywoływana dla obiektu graficznego utworzonego dla panelu poprzez polecenie ***pole_gry.CreateGraphics().DrawString***. Funkcja ta w parametrach przyjmuje kolejno: **A** – tekst, jaki chcemy utworzyć; **B** – obiekt typu **Font**, **C** – obiekt typu **Brush**, **D** – współrzędną **x**, z której zostanie utworzony tekst, **E** – współrzędną **y** tekstu.

```
public Form1()
{
    InitializeComponent();
    czy_gra_aktywna = false;
    timer1.Enabled = true;
```

9 Jeśli teraz uruchomimy grę, napis nie zostanie wyświetlony. A to dlatego, że zdarzenie **Tick** dla **Timera** nie jest wywoływane. Aby mogło ono występować, właściwość **Enabled** dla **Timera** musi mieć wartość **true**, a domyślnie ma wartość **false**. Przypiszemy tej właściwości wartość **true** w konstruktorze **Form1** po nadaniu wartości ***czy_gra_aktywna***. Po dodaniu brakującej linii, gdy włączymy podgląd, zobaczymy już napis.



Startowanie gry

Wiemy już, co ma być wyświetlane, gdy gra nie jest aktywna. A co powinno się dziać, gdy gra będzie już aktywowana?

1 Rozpoczynamy od nadania ***czy_gra_aktywna*** wartości **true**. Możemy to zrobić w metodzie wywoływanej po kliknięciu na **Start**. W tym celu przechodzimy do okna modelowania i dwukrotnie klikamy na przycisk **Start** z menu głównego, aby wygenerować metodę. W jej treści aktywujemy grę.

```
private void startToolStripMenuItem_Click(object sender, EventArgs e)
{

}
```

```
private void startToolStripMenuItem_Click(object sender, EventArgs e)
{
    czy_gra_aktywna = true;
}
```

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
        pole_gry.CreateGraphics().Clear(Color.Black);
    }
    else
    {
        FontFamily fontFamily1 = new FontFamily("Arial");
        Font f = new Font(fontFamily1, 15);
        Brush b = new SolidBrush(Color.Aqua);
        pole_gry.CreateGraphics().DrawString("Naciśnij Start", f, b, 80, 135);
    }
}
```

2 Zmiana wartości ***czy_gra_aktywna*** spowoduje przejście do innej części instrukcji **if** w metodzie odpowiedzialnej za działanie **Timera**.

Dla nas nie będzie to widoczna różnica, ponieważ w tym momencie nie wykona się nic nowego, a na ekranie po kliknięciu na przycisk **Start** nadal będzie widać ten tekst. A zatem w instrukcji **if**, kiedy podany przez nas warunek jest spełniony, wyczyśćmy **pole_gry**, aby umieszczony napis przestał być widoczny.

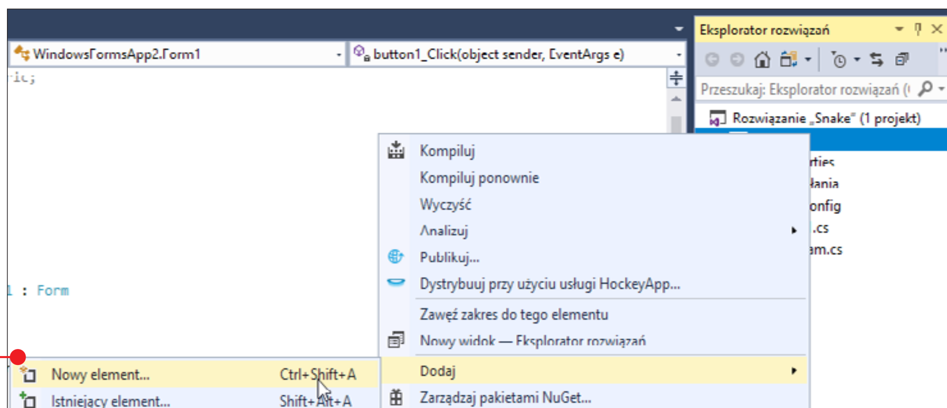
Zrobimy to poleceniem ***pole_gry.CreateGraphics().Clear()***, gdzie w parametrze funkcji podamy kolor, jakim ma być wypełniony panel **pole_gry**. Jeśli teraz przetestujemy grę, zobaczymy, że napis zniknie po kliknięciu na przycisk **Start**.

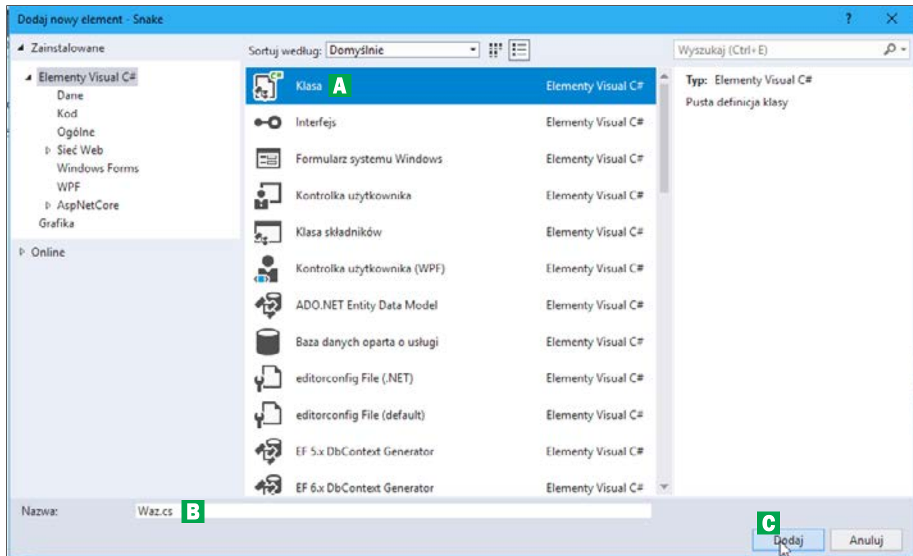
Klasa Waz

Obiektowe spojrzenie na programowanie wymaga umiejętności wychwycenia, jakie obiekty budują naszą grę. W grze Snake są to poruszający się po planszy wąż, a także owoce pojawiające się w losowych miejscach, które wąż ma zjadać.

Aby móc stworzyć obiekty – węża i owoce – będziemy musieli napisać stosowne klasy. Treść klas umieścimy w oddzielnych plikach.

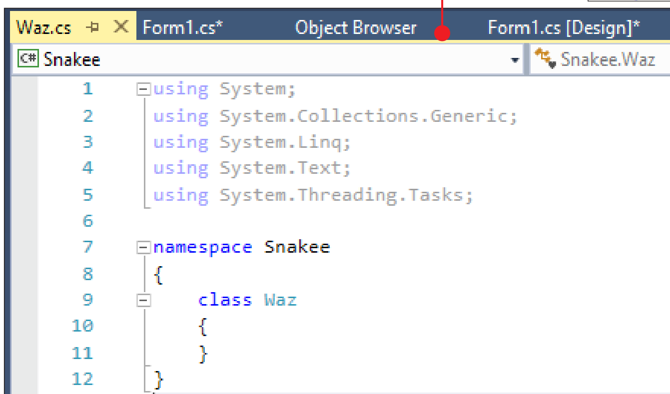
1 Aby utworzyć taki plik z klasą, klikamy prawym przyciskiem na nazwę projektu w **Eksploratorze rozwiązań** widocznym po prawej stronie okna narzędzia. Następnie z menu kontekstowego wybieramy **Dodaj**, a z rozwiniętej listy – **Nowy element**.





2 W nowym oknie wybieramy typ pliku – **Klasa** **A**. W pole **Nazwa** wpisujemy nazwę klasy – **Waz.cs** **B** (chodzi o węża, ale w nazwach klas nie możemy używać polskich znaków) i tworzymy ją, klikając na przycisk **Dodaj** **C**.

3 Mamy teraz plik z podstawową strukturą i miejscem na definicję klasy **Waz** ●.



4 Wąż będzie składał się z segmentów – czyli kwadratów. Ich liczba będzie rosła z każdym zjedzonym przez węża owocem. Liczbę segmentów zapiszemy jako pole **segmenty** klasy **Waz**. Ponieważ liczba segmentów jest liczbą całkowitą, użyjemy typu danych **Integer** ●.

```
class Waz
{
    public int segmenty;
```

```
class Waz
{
    public int segmenty;
    public int segment;
```

5 Stworzymy też pole **segment**, również typu **Integer** ●, którego wartość będzie wielkością jednego segmentu węża.

6 Każdy z segmentów węża będzie miał inną lokalizację. Lokalizacja każdego z segmentów określona jest współrzędnymi **x** i **y**. Tworzymy zatem dwie tablice ● – jedną dla współ-

```
class Waz
{
    public int segmenty;
    public int segment;
    public int[] x = new int[900];
    public int[] y = new int[900];
    public string ruch;
```

rzędnych **x**, drugą dla współrzędnych **y**. Pod indeksem **0** będą mieścić się w tablicach współrzędne głowy węża. Pod kolejnymi indeksami – kolejne segmenty węża.

```
class Waz
{
    public int segmenty;
    public int segment;
    public int[] x = new int[900];
    public int[] y = new int[900];
    public string ruch;
```

7 Ostatnim polem w klasie będzie **ruch**. Powinien on mieć typ **string**. Ruch będzie mógł przybierać wartości **prawo**, **lewo**, **gora** i **dol** – zależnie od jego wartości będziemy przesuwać węża w odpowiednią stronę.

Konstruktor klasy Waz

Czas napisać konstruktor dla klasy **Waz**.

1 Deklarujemy go w taki sposób, aby tworząc obiekt klasy **Waz**, podawać w parametrze konstruktora szerokość i wysokość pola, na którym rozgrywana będzie gra.

```
public Waz(int szerokosc, int wysokosc)
{

}
```

2 Szerokość wykorzystamy do obliczenia wielkości jednego segmentu. Aby w poprzek pola gry mogło zmieścić się 20 segmentów węża, wielkość jednego segmentu powinna być taka jak 1/20 szerokości pola gry. Dlatego do wartości **segment** przypisujemy **szerokosc** podzieloną na **20**.

```
public Waz(int szerokosc, int wysokosc)
{
    segment = szerokosc / 20;
```

3 Określimy też początkową liczbę segmentów węża. W momencie rozpoczęcia gry wystarczą trzy segmenty.

```
public Waz(int szerokosc, int wysokosc)
{
    segment = szerokosc / 20;
    segmenty = 3;
```

4 Przypisując do ruchu wartość **prawo**, określamy, że na początku gry wąż będzie poruszał się w prawą stronę.

```
public Waz(int szerokosc, int wysokosc)
{
    segment = szerokosc / 20;
    segmenty = 3;
    ruch = "prawo";
```

5 Aby ustawić węża tak, żeby jego głowa była na środku pola gry, wprowadzamy dwie lokalne zmienne – **xg** i **yg**, które posłużą nam do obliczenia współrzędnych głowy węża. Współrzędne określają zawsze lewy górny róg kwadratu. Jeśli w szerokości pola gry mieści się 20 segmentów, ustalimy, że dziesiątym segmentem będzie głowa. Segment dziesiąty zaczyna się tam, gdzie kończy dziewiąty – zatem współrzędna **x** dziesiątego segmentu jest równa sumie dziewięciu szerokości segmentów. Do wartości **xg** przypisujemy szerokość segmentu pomnożoną przez 9. Podobnie możemy zrobić z **yg**.

```
segment = szerokosc / 20;
segmenty = 3;
ruch = "prawo";
int xg = 9 * segment;
int yg = 9 * segment;
for (int i = 0; i < segmenty; i++)
{
    x[i] = xg - (i * segment);
    y[i] = yg;
}
```

6 Dalej tworzymy pętlę **for**, w której będziemy przypisywać odpowiednie wartości współrzędnych **x** i **y** w tablicach, dla poszczególnych segmentów węża. Pętla **for** iteruje od zmiennej **i** równej zero. Jeżeli do **x[0]** przypiszemy **xg**, to do kolejnych wartości **x[i]** będziemy musieli przypisywać wartość o jeden segment mniejszą niż wcześniej. Jeżeli w pętli **for** będziemy mnożyć **i** razy szerokość segmentu i odejmować to od **xg (segment dziesiąty)**, to dla **x[0]**, pomimo że odejmujemy (**i*segment**) od **xg**, uzyskamy wartość **xg**, gdyż **i** jest równe **0**, zatem **0*segment** też będzie równe **0**. Jest to dla nas idealne rozwiązanie.

```
segment = szerokosc / 20;
segmenty = 3;
ruch = "prawo";
int xg = 9 * segment;
int yg = 9 * segment;
for (int i = 0; i < segmenty; i++)
{
    x[i] = xg - (i * segment);
    y[i] = yg;
}
```

7 Dla **y[i]** będziemy przypisywać tę samą wartość **yg** – to dlatego, że wąż ma być ustawiony poziomo. Wtedy wszystkie jego segmenty będą początkowo miały taką samą wartość współrzędnej **y**.

Ruch węża

W klasie **Waz** znajdzie się także odpowiednia metoda, która będzie odpowiedzialna za poruszanie wężem. Jedno wywołanie metody będzie przesuwało głowę węża w kierunku ustalonym przez wartość pola **ruch**, a każdy segment węża będzie przesuwany w miejsce segmentu, który go poprzedza. Tak zbudowana metoda będzie mogła być wywoływana poprzez **Timer**.

1 Deklarujemy w klasie **Waz** metodę **move** w typie **void** (nie będzie zwracać danych).

```
public void move()
{

}
```

2 Nie możemy zacząć przesuwać segmentów węża od pierwszego jego elementu (od głowy). Jeśli przesuniemy głowę, to utracimy informację, gdzie znajdowała się głowa i nie będziemy wiedzieć, w jakim miejscu ustawić segment węża pierwszy po głowie. Dlatego przesuwanie węża zaczynamy od jego ostatniego elementu i przemieszczamy go w miejsce elementu przedostatniego. Podobnie z elementem przedostatnim – który przesuwamy w miejsce poprzedzającego go elementu – i tak dalej. Przesuwanie segmentów węża wykonamy poprzez pętlę **for**. Iterująca w pętli zmienna **i** będzie początkowo przyjmować wartość liczby segmentów węża. Z każdym przejściem pętli będzie zmniejszać ona swoją wartość aż do zera – kiedy to pętla przestanie się wykonywać. We wnętrzu pętli zmieniamy wartość współrzędnych **x** i **y** segmentu o indeksie **i** na wartość współrzędnych segmentu o indeksie o jeden mniejszego od **i**.

```
public void move()
{
    for (int i = segmenty; i > 0; i--)
    {
        x[i] = x[(i - 1)];
        y[i] = y[(i - 1)];
    }
}
```

```
public void move()
{
    for (int i = segmenty; i > 0; i--)
    {
        x[i] = x[(i - 1)];
        y[i] = y[(i - 1)];
    }
A   if (ruch == "lewo")
    {
        x[0] = x[0] - segment;
    }
B   if (ruch == "prawo")
    {
        x[0] = x[0] + segment;
    }
C   if (ruch == "gora")
    {
        y[0] = y[0] - segment;
    }
C   if (ruch == "dol")
    {
        y[0] = y[0] + segment;
    }
```

3 Ostatnie wykonanie pętli **for** to przeniesienie segmentu o indeksie **1** w miejsce segmentu o indeksie **0** (czyli głowy). Pętla **for** nie przenosi głowy węża. Głowę przeniesiemy w dalszej części metody **move**, zmieniając jej współrzędną **x** lub **y** zależnie od tego, jaka jest wartość pola **ruch**. Punkt, w którym **x** i **y** mają wartości **0**, znajduje się w lewym górnym rogu pola gry. Wartości na osi **y** rosną do dołu okna gry, a na osi **x** rosną w prawą stronę. Kiedy mamy poruszać się w stronę lewą **A**, musimy zmniejszyć wartość współrzędnej **x** o wielkość jednego segmentu. Jeśli w stronę prawą – zwiększamy wartość współrzędnej **x** **B** głowy węża o wielkość segmentu. Analogicznie robimy ze współrzędną **y** głowy węża, gdy chcemy poruszać się w górę i dół **C**.

```
private void startToolStripMenuItem_Click(object sender, EventArgs e)
{
    czy_gra_aktywna = true;
    snake = new Waz(pole_gry.Width, pole_gry.Height); E
}           D
```

4 Wąż nie może przemieszczać się bez ograniczeń. Kiedy wyjdzie poza prawą krawędź pola gry, powinnien pojawić się z lewej strony. Gdy wyjdzie poza lewy brzeg – pojawi się z prawej. Analogicznie powinno się dziać, gdy wąż będzie przekraczał górną i dolną krawędź pola gry. Jak to zrobić? W metodzie **move** musimy sprawdzać, czy nadal mieścimy się na polu gry. Jeśli współrzędna **x** głowy jest mniejsza niż najmniejsza możliwa współrzędna, czyli **0**, powinniśmy przenieść głowę do takiej współrzędnej **x**, która jest największą możliwą współrzędną, aby głowa była widoczna na polu gry. Jeśli

```
    }
    if (ruch == "dol")
    {
        y[0] += segment;
    }
    if (x[0] < 0) { x[0] = segment * 19; }
    if (x[0] > segment * 20) { x[0] = 0; }
    if (y[0] < 0) { y[0] = segment * 19; }
    if (y[0] > segment * 20) { y[0] = 0; }
}
```

pole gry mieści 20 segmentów, to ostatni segment ma współrzędną **x** równą dziewiętnastokrotności szerokości segmentu.

Obiekt klasy Waz

Stworzyliśmy już dużą część klasy **Waz**. Aby sprawdzić działanie stworzonej przez nas metody **move**, musimy teraz utworzyć obiekt napisanej przez nas klasy i dla tego obiektu wywołać naszą metodę. Powracamy zatem do pliku **Form1.cs** i w nim kontynuujemy pisanie.

```
namespace Snake
{
    public partial class Form1 : Form
    {
        private bool czy_gra_aktywna;
        private Waz snake;
```

1 Deklarujemy dla klasy **Form1** pole będące obiektem klasy **Waz** – możemy nazwać je **snake**.

2 Przenosimy się do metody odpowiedzialnej za kliknięcie na przycisk startujący grę. W niej dla zadeklarowanego wyżej pola klasy wywołujemy konstruktor D. Przyjmuje on w swoich parametrach szerokość i wysokość pola, na którym rozgrywana będzie gra. My mamy obiekt **pole_gry**. Jego wielkość ustawiliśmy na 280 x 280 pikseli. Nie będziemy jednak wpisywać w tym miejscu tych samych liczb. Programując, staramy się unikać takich sytuacji. Gdybyśmy w przyszłości stwierdzili, że chcemy zmienić rozmiar pola gry, musielibyśmy zmieniać liczbę odpowiadającą jego rozmiarowi w różnych miejscach kodu. Zamiast tego możemy wpisać polecenie, które samo odczyta wymiary obiektu **pole_gry** – niezależnie od tego, jakie one są. Zatem w pierwszym parametrze konstruktora obiektu **snake** piszemy **pole_gry.Width**, a w drugim **pole_gry.Height** E.

3 Moglibyśmy już wywołać metodę **move** dla obiektu **snake**. Jej wywołanie powinno znajdować się w metodzie odpowiedzialnej za działanie **Timera**.

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
        pole_gry.CreateGraphics().Clear(Color.Black);
        snake.move();
```

4 Jeśli teraz uruchomimy grę i klikniemy na przycisk **Start**, nasz wąż będzie się przemieszczał, ale… tylko wirtualnie. To dlatego, że nie napisaliśmy jeszcze żadnej metody odpowiedzialnej za „zmaterializowanie się” węża na polu gry. Wracamy zatem do definicji klasy **Waz**.

```
public void rysuj(Graphics g, Brush b)
{
    for (int i = 0; i < segmenty; i++)
    {

    }
}
```

```
public void rysuj(Graphics g, Brush b)
{
    for (int i = 0; i < segmenty; i++)
    {
        g.FillRectangle(b, x[i], y[i], segment, segment);
    }
}
```

A B C D E

W niej tworzymy metodę **rysuj**. Będzie ona wyrysowywać kwadraty w miejscu każdego segmentu węża. Rysowanie jest możliwe tylko wtedy, gdy mamy obiekt graficzny i pędzel. Dlatego nasza metoda **rysuj** będzie wymagała podania dwóch parametrów: obiektu **g** typu **Graphics** oraz obiektu **b** typu **Brush** – są one niezbędne do rysowania.

```
public void rysuj(Graphics g, Brush b)
{

}
```

5 Program po wpisaniu takich parametrów podkreśla błędy. To dlatego, że te elementy znajdują się w przestrzeni **System.Drawing**. Musimy zatem dopisać u góry klasy **Waz**: **using System.Drawing;** – to usunie błędy.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;
```

```
public void rysuj(Graphics g, Brush b)
{

}
```

6 Dalej, w treści metody **rysuj**, musimy stworzyć pętlę **for**, która wykona się dla wszystkich elementów węża.

7 Wewnątrz pętli **for** dla obiektu graficznego **g** wywołujemy metodę **FillRectangle**, która pozwala wypełnić kolorem prostokąt – a w jej parametrach musimy podać kolejno:

- **A** pędzel, którym będziemy rysować
- **B** współrzędną **x** lewej krawędzi prostokąta
- **C** współrzędną **y** górnej krawędzi prostokąta
- **D** szerokość rysowanego prostokąta
- **E** wysokość rysowanego prostokąta

Ponieważ instrukcja jest wykorzystana w pętli, możemy odnosić się po kolei do wszystkich elementów tablic **x[]** i **y[]**. Jako szerokość i wysokość prostokąta wykorzystamy wartość pola **segment** – nasz wąż będzie składał się z kwadratów.

8 Tak napisaną metodę **rysuj** należy wywołać dla obiektu **snake**. Powracamy zatem do **Form1.cs** i tam kontynuujemy pisanie. Przenosimy się do metody odpowiedzialnej za zachowanie **Timera**. To w niej, po każdym przesunięciu obiektu **snake**, powinniśmy narysować węża w nowym miejscu. Piszemy zatem **snake.rysuj()** **A**. Pamiętajmy jednak o parametrach tej metody. Jako pierwszy parametr musimy podać obiekt graficzny – tworzymy go z obiektu **pole_gry** poprzez polecenie ***pole_gry.CreateGraphics()*** **B**. Drugim parametrem jest pędzel – który utworzymy bezpośrednio w wywołaniu metody **rysuj** poprzez konstruktor **SolidBrush** **C** wymagający od nas podania jedynie koloru, jakim chcemy rysować.

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
        pole_gry.CreateGraphics().Clear(Color.Black);
        snake.move();
        snake.rysuj(pole_gry.CreateGraphics(), new SolidBrush(Color.Aqua));
```

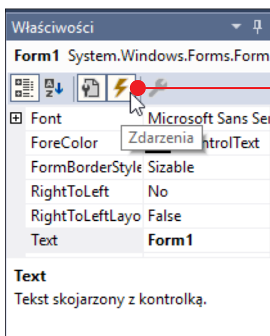
A B C

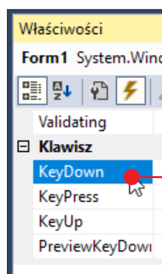
9 Gdy teraz uruchomimy naszą grę, zobaczymy, że wąż porusza się w prawą stronę i wyjeżdża poza prawą krawędź okna, po czym powraca z lewej strony pola gry.

Sterowanie wężem

Stworzony przez nas wąż potrafi się przemieszczać, jednak nie zmienia kierunku. Mimo że w metodzie **move** klasy **Waz** napisaliśmy fragment odpowiadający za to, żeby nasz wąż mógł zmieniać kierunek, nigdzie nie zmieniamy jeszcze wartości jego pola **ruch**.

1 Przejdźmy teraz do podglądu modelowanego okna gry. Tam znajduje się znana nam sekcja **Właściwości**, w której do tej pory zmienialiśmy właściwości elementów budujących nasz program. Teraz, mając zaznaczone okno gry (**Form1**), klikamy na przycisk z piorunem (**Zdarzenia**).





2 Przeniesieni zostaliśmy do listy zdarzeń, jakie mogą być wywołane dla okna naszej gry. Odszukujemy zdarzenie **KeyDown** i klikamy na nie dwukrotnie.

3 W klasie **Form1** zostanie wygenerowana metoda, która będzie wywoływana w momencie naciśnięcia dowolnego klawisza z klawiatury.

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{

}
```

4 Metoda jest wywoływana naciśnięciem każdego klawisza z klawiatury, jednak nas interesować będą tylko cztery klawisze – strzałki. Za pomocą czterech instrukcji warunkowych będziemy sprawdzać, czy kod

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Up) snake.ruch = "gora";
    if (e.KeyCode == Keys.Down) snake.ruch = "dol";
    if (e.KeyCode == Keys.Right) snake.ruch = "prawo";
    if (e.KeyCode == Keys.Left) snake.ruch = "lewo";
}
```

naciśniętego klawisza odpowiada kolejno strzałkom w górę, dół, prawo i lewo. Następnie, gdy któryś warunek zostanie spełniony, zmienimy wartość przypisaną do pola **ruch** w obiekcie **snake** na taką, która zmieni jego kierunek zależnie od naciśniętej strzałki.

Owoc – tworzymy i wyświetlamy

Nasz wąż może się już poruszać, brakuje mu jedynie celu, w kierunku którego powinien pełzać. Musimy więc stworzyć owoc, który ma być celem węża. Owoc utworzymy jako obiekt klasy **Owoc**, którą sami napiszemy.

1 Tworzymy nowy plik z klasą **Owoc.cs**, powtarzając czynności, które wykonaliśmy, tworząc plik **Waz.cs**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Snake
{
    class Owoc
    {
    }
}
```

2 W klasie **Owoc** deklarujemy trzy pola, wszystkie w typie **Integer**. Pierwsze to współrzędna **x** **A** lewej krawędzi owocu, drugie to współrzędna **y** **B** górnej krawędzi owocu. Trzecie pole – o nazwie **segment** **C**,

```
class Owoc
{
    public int x; A
    public int y; B
  C public int segment;
}
```

będzie wielkością jednego segmentu planszy – znajdzie się tam taka sama wartość jak w wężu, aby owoc był wielkości jednego segmentu węża.

3 Zanim przejdziemy do tworzenia konstruktora klasy **Owoc**, stworzymy metodę ***nowy_owoc***, która będzie odpowiedzialna za generowanie owocu w losowym miejscu. Metoda ta powinna zawierać generator liczb losowych – obiekt typu **Random** (poznaliśmy go już przy okazji tworzenia gry Wisielec). Dzięki niemu możemy nadać pseudolosowe wartości polom **x** i **y**. Nie mogą to być jednak dowolne losowe wartości z przedziału od jednej do drugiej krawędzi okna. Mogą to być tylko i wyłącznie wartości odpowiadające tym miejscom, w których może znaleźć się głowa węża. To znaczy, że pomiędzy kolejnymi możliwymi wartościami musi być różnica równa wielkości jednego segmentu. Aby uzyskać taki efekt, losujemy wartość z przedziału 0–20 (funkcja **Next** daje nam liczbę losową, która może być równa lub większa od dolnej granicy losowania i jednocześnie mniejsza niż górna granica – podanie takiego zakresu pozwala na wylosowanie liczb od 0 do 19).

```
public void nowy_owoc()
{
    Random r = new Random();
    x = r.Next(0, 20) * segment;
    y = r.Next(0, 20) * segment;
}
```

4 Metodę ***nowy_owoc*** stworzyliśmy jeszcze przed napisaniem konstruktora, ponieważ będziemy wywoływać metodę ***nowy_owoc*** w konstruktorze. Deklarujemy teraz konstruktor dla klasy **Owoc**. Będzie on w parametrze przyjmował wielkość segmentu, którą przypisujemy do pola **segment**.

```
public Owoc(int segment)
{
    this.segment = segment;
    nowy_owoc();
}
```

Tuż po przypisaniu wywołujemy metodę ***nowy_owoc***, aby wartości dostały też pozostałe dwa pola klasy.

5 Nie jest to jedyne miejsce w kodzie klasy **Owoc**, w którym wywołamy metodę ***nowy_owoc***. Metoda ta będzie wywołana również w innej metodzie – tej, w której będziemy sprawdzać, czy należy wygenerować nowy owoc. Generowanie nowego owocu jest konieczne, gdy wąż zje ten, który aktualnie jest widoczny. Tworzymy metodę ***czy_nowy_owoc***. Tym razem będzie to metoda typu boolowskiego – jej wywołanie będzie zwracać wartość **true** lub **false** zależnie od tego, czy należy wygenerować nowy owoc. Aby móc stwierdzić, czy owoc i głowa węża są w tym samym miejscu, metoda powinna pobierać w parametrze dwie wartości odpowiadające współrzędnym **x** i **y** głowy węża.

```
public bool czy_nowy_owoc(int glowa_x, int glowa_y)
{

}
```

6 Następnie tworzymy instrukcję warunkową **if**, która sprawdza, czy **x** owocu jest takie samo jak **x** głowy węża i jednocześnie, czy **y** owocu jest takie samo jak **y** głowy węża. Jeśli tak – metoda powinna zwracać prawdę (**true**) i wywołać kolejne wykonanie metody ***nowy_owoc***, aby nadać nową losową lokalizację owocu. Jeśli warunek nie jest spełniony – metoda powinna zwracać wartość **false**.

```
public bool czy_nowy_owoc(int glowa_x, int glowa_y)
{
    if (x == glowa_x && y == glowa_y)
    {
        nowy_owoc();
        return true;
    }
    return false;
}
```

7 Na nic nam jednak nie zda się generowanie nowych owoców, jeśli nie będziemy widzieć ich na polu gry. W klasie **Owoc** powin-

```
public void rysuj_owoc(Graphics g, Brush b)
{
    g.FillRectangle(b, x, y, segment, segment);
}
```

na także znaleźć się metoda ***rysuj_owoc***, której treść powinna być zbudowana analogicznie do metody **rysuj** z klasy **Waz**, tyle że bez pętli **for**, ponieważ tu do narysowania mamy tylko jeden prostokąt. Podobnie jak w przypadku rysowania węża – pamiętajmy, że aby wszystko działało poprawnie, powinniśmy dodać polecenie ***using System.Drawing;*** u góry klasy.

8 Pora wykorzystać napisaną przez nas klasę. Wracamy do pliku **Form1.cs** i w nim deklarujemy pole **owoc** w typie **Owoc**.

```
namespace Snake
{
    public partial class Form1 : Form
    {
        private bool czy_gra_aktywna;
        private Waz snake;
        private Owoc owoc;
        public Form1()
```

9 Następnie w metodzie odpowiedzialnej za kliknięcie na przycisk **Start** dopisujemy wywołanie konstruktora klasy **Owoc**. W jego parametrze musimy podać wielkość segmentu planszy – jej wartość możemy odczytać z obiektu **snake**.

Zjadanie owocu

W klasie **Owoc** mamy już metodę, która sprawdza, czy należy wygenerować nowy owoc. Zanim z niej skorzystamy, napiszemy jeszcze inną metodę – w klasie **Waz**. Będziemy ją wywoływać, kiedy owoc zostanie zjedzony, aby wąż mógł urosnąć.

1 Tworzymy w klasie **Waz** metodę ***dodaj()***. Będzie ona tworzyć nowe elementy w tablicach **x[]** i **y[]** – pod indeksem odpowiadającym liczbie segmentów. Pamiętajmy, że elementy w tablicach indeksowane są od zera, a nie od jedynki. Jeśli pole **segmenty** ma

```
public void dodaj()
{
    x[segmenty] = x[segmenty - 1];
    y[segmenty] = y[segmenty - 1];
    segmenty = segmenty + 1;
}
```

wartość **5**, to znaczy, że mamy w tablicach elementy o indeksach od **0** do **4**. Kiedy dodamy element o indeksie **5** – automatycznie zwiększymy rozmiar tablicy, czyli zwiększymy liczbę segmentów – dlatego powinniśmy na koniec definicji tej metody powiększyć o jeden wartość pola **segmenty**.

2 Wracamy do **Form1.cs** i w metodzie odpowiedzialnej za zachowanie **Time-**

```
private void startToolStripMenuItem_Click(object sender, EventArgs e)
{
    czy_gra_aktywna = true;
    snake = new Waz(pole_gry.Width, pole_gry.Height);
    owoc = new Owoc(snake.segment);
```

10 W metodzie odpowiadającej za działanie **Timera** po rysowaniu węża uruchamiamy także rysowanie owocu.

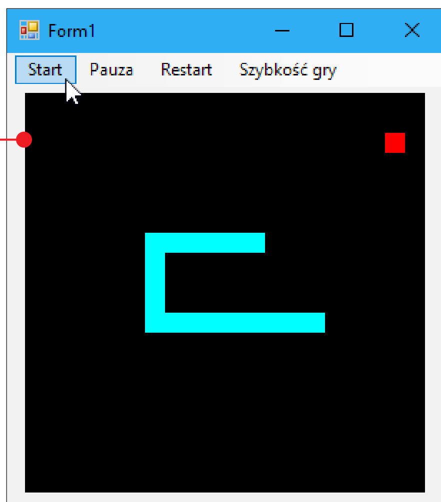
ra po wywołaniu rysowania owocu tworzymy instrukcję warunkową **if**. W jej warunku umieszczamy wywołanie metody

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
        pole_gry.CreateGraphics().Clear(Color.Black);
        snake.move();
        snake.rysuj(pole_gry.CreateGraphics(), new SolidBrush(Color.Aqua));
        owoc.rysuj_owoc(pole_gry.CreateGraphics(), new SolidBrush(Color.Red));
```

```
if (czy_gra_aktywna)
{
    pole_gry.CreateGraphics().Clear(Color.Black);
    snake.move();
    snake.rysuj(pole_gry.CreateGraphics(), new SolidBrush(Color.Aqua));
    owoc.rysuj_owoc(pole_gry.CreateGraphics(), new SolidBrush(Color.Red));
    if (owoc.czy_nowy_owoc(snake.x[0], snake.y[0]))
    {
        snake.dodaj();
    }
}
```

czy_nowy_owoc dla obiektu **owoc**. Jeśli zwróci ona wartość **true** – czyli gdy warunek zostanie spełniony – należy wykonać metodę ***dodaj*** dla obiektu **snake**.

3 Jeśli teraz przetestujemy grę, mamy już węża, który może poruszać się we wszystkich czterech możliwych kierunkach i rosnąć po zjedzeniu owocu. Tylko że... Możemy teraz grać i grać bez końca, bo nasza gra nie ma napisanego zakończenia!



Programujemy przegraną

Aby nasza gra miała sens, powinniśmy nie tylko mieć możliwość niekontrolowanego rośnięcia, ale także unikać zjedzenia węża przez samego siebie. Gdy głowa węża znajdzie się w miejscu jakiegokolwiek innego jego segmentu, powinniśmy ogłosić przegraną.

1 Przechodzimy do pliku **Waz.cs**, w którym mamy definicję klasy **Waz**. Dodajemy tam metodę ***czy_waz_zyje*** – ona również będzie typu boolowskiego.

```
public bool czy_waz_zyje()
{
    for (int i = 1; i < segmenty; i++)
    {
        if (x[0] == x[i] && y[0] == y[i])
        {
            return false;
        }
    }
    return true;
}
```

2 Metoda ta składać się powinna z pętli **for**, która pozwoli nam na odnoszenie się do kolejnych segmentów węża – oprócz jego głowy. To właśnie głowę – a raczej jej współrzędne, będziemy poprzez instrukcję warunkową **if** porównywać ze współrzędnymi kolejnych segmentów. Jeśli któryś z segmentów pokrywa się lokalizacją z głową, metoda powinna zwrócić **false**. Jeżeli metoda przejdzie przez całą pętlę bez zwrócenia po drodze **false**, powinna na koniec zwrócić **true**.

3 Wracamy do **Form1.cs** – tam w metodzie odpowiedzialnej za zachowanie **Timera** musimy stworzyć instrukcję warunkową **if**, która będzie sprawdzać, co zwraca stworzona uprzednio dla obiektu **snake** metoda ***czy_waz_zyje***. Jeśli zwróci ona **false**, zmieniamy wartość pola ***czy_gra_aktywna***, co zatrzyma naszą grę.

```
private void startToolStripMenuItem_Click(object sender, EventArgs e)
{
    czy_gra_aktywna = true;
    snake = new Waz(pole_gry.Width, pole_gry.Height);
    owoc = new Owoc(snake.segment);
    pauzaToolStripMenuItem.Enabled = true;
}
```

Programujemy pozostałe przyciski menu

Nasza gra działa już poprawnie, jednak w naszym menu głównym mamy jeszcze kilka przycisków, na które kliknięcie podczas gry nic nam nie daje, ponieważ ich jeszcze nie zaprogramowaliśmy.

1 Mamy przycisk **Pauza** – kliknięcie na niego powinno wstrzymać grę. Nie możemy wstrzymać gry, jeśli nie została ona jeszcze rozpoczęta. Dlatego przycisk ten początkowo powinien być nieaktywny. Przechodzimy do konstruktora **Form1** i dodajemy tam polecenie ***pauzaToolStripMenuItem.Enabled = false;*** – jego działanie spowoduje brak możliwości kliknięcia na przycisk **Pauza**.

2 Przycisk **Pauza** powinien być ponownie aktywny, gdy klikniemy na **Start** i uruchomimy grę. Przechodzimy do metody odpowiedzialnej za kliknięcie na przycisk **Start** i dodajemy tam ***pauzaToolStripMenuItem.Enabled = true;***.

```
public Form1()
{
    InitializeComponent();
    czy_gra_aktywna = false;
    timer1.Enabled = true;
    pauzaToolStripMenuItem.Enabled = false;
}
```

3 Wracamy do modelowania okna i dwukrotnie klikamy na przycisk **Pauza**, aby wygenerować metodę odpowiedzialną za kliknięcie na niego. W jej treści tworzymy instrukcję warunkową **if**, która sprawdza wartość pola **czy_gra_aktywna**. To dlatego, że przycisk ten inaczej powinien działać, gdy chcemy zatrzymać grę – a inaczej, gdy chcemy ją wznowić. Jeśli gra jest aktywna – należy ją za-

```
private void pauzaToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
        czy_gra_aktywna = false;
        pauzaToolStripMenuItem.Text = "Wznów";
        pole_gry.CreateGraphics().Clear(Color.Black);
    }
```

```
if (czy_gra_aktywna)
{
    pole_gry.CreateGraphics().Clear(Color.Black);
    snake.move();
    snake.rysuj(pole_gry.CreateGraphics(), new SolidBrush(Color.Aqua));
    owoc.rysuj_owoc(pole_gry.CreateGraphics(), new SolidBrush(Color.Red));
    if (owoc.czy_nowy_owoc(snake.x[0], snake.y[0]))
    {
        snake.dodaj();
    }
    if (snake.czy_waz_zyje() == false)
    {
        czy_gra_aktywna = false;
    }
```

```
private void pauzaToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
        czy_gra_aktywna = false; A
        pauzaToolStripMenuItem.Text = "Wznów"; B
        pole_gry.CreateGraphics().Clear(Color.Black);
    }
    else C
    {
        czy_gra_aktywna = true;
        pauzaToolStripMenuItem.Text = "Pauza";
    }

}
```

trzymać. To znaczy, że zmieniamy wartość pola **czy_gra_aktywna** na **false** A. Możemy również zmienić napis na przycisku, modyfikując jego wartość właściwości **Text** na **Wznów** B. Dodatkowo – po tym poleceniu możemy użyć funkcji, która wyczyści panel **pole_gry**.

4 Kiedy gracz ponownie kliknie na przycisk **Pauza** – gdy gra nie jest aktywna (sekcja **else** C naszej instrukcji warunkowej), powinniśmy ją aktywować poprzez nadanie polu **czy_gra_aktywna** wartości **true**. Dodatkowo powinniśmy powrócić do pierwotnego napisu na przycisku, zmieniając jego właściwość **Text** na **Pauza**.

5 Powracamy do okna modelowania i generujemy metodę odpowiedzialną za kliknięcie na przycisk **Restart**. Kliknięcie na niego powinno wygenerować nowego węża oraz nowy owoc – właśnie to piszemy, wywołując konstruktory naszych obiektów.

```
private void restartToolStripMenuItem_Click(object sender, EventArgs e)
{
    snake = new Waz(pole_gry.Width, pole_gry.Height);
    owoc = new Owoc(snake.segment);
}
```

6 Dodatkowo polecenia te możemy zamknąć w instrukcję warunkową **if** – ponieważ restart gry powinien być możliwy oczywiście tylko w takim wypadku, gdy została ona wcześniej uruchomiona. Sprawdźmy zatem, czy gra jest aktywna i tylko w takiej sytuacji wywołujmy ponownie konstruktory.

```
private void restartToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
        snake = new Waz(pole_gry.Width, pole_gry.Height);
        owoc = new Owoc(snake.segment);
    }
}
```

7 Wracamy do modelu okna gry i generujemy metody odpowiedzialne za kliknięcia przycisków **Wolniej** i **Szybciej**. Ich treść będzie podobna, ponieważ opierać się będzie ona na zmianie wartości właściwości **Interval** dla naszego **Timera** (interwał to odstęp pomiędzy kolejnymi wywołaniami metody odpowiedzialnej za „tyknięcia" **Timera**). Im większy interwał, tym dłuższy czas. Im dłuższy czas – tym wolniej działa nasza gra. Po kliknięciu na przycisk **Wolniej** powinniśmy dodać 10 milisekund do interwału. Kiedy zaś klikniemy na przycisk **Szybciej**, powinniśmy odjąć 10 milisekund od

```
private void wolniejToolStripMenuItem_Click(object sender, EventArgs e)
{
    timer1.Interval += 10;
}

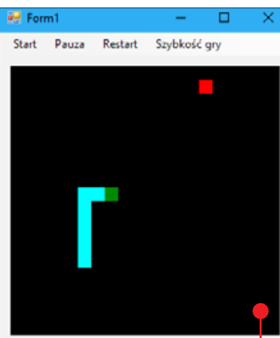
private void szybciejToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (timer1.Interval > 10) { timer1.Interval -= 10; }
}
```

interwału. W tym jednak przypadku nie możemy zejść z wartością interwału poniżej zera! Dlatego gdy odejmujemy 10 milisekund, to interwał musi być większy niż 10 milisekund – aby pozostała tam jakakolwiek wartość większa od zera.

DLA AMBITNYCH: GŁOWA WĘŻA

Rysując węża, stworzyliśmy go w taki sposób, że jest on cały w jednym kolorze. W tego typu grach twórcy często nadają głowie węża inny kolor, aby była ona lepiej widoczna. My też możemy tak zrobić. W tym celu należy zmodyfikować metodę **rysuj** w klasie **Waz**. Przed pętlą **for** należy napisać instrukcję, która narysuje głowę węża w określonym przez nas kolorze. Samą pętlę **for** – musimy zmienić tak, aby w jej wnętrzu rysowane były wszystkie inne segmenty węża oprócz głowy, czyli pętla powinna iterować na zmiennej **i**, króra na początek otrzyma wartość **1**, a nie **0**. A oto efekt z głową wyróżnioną kolorem.

```
public void rysuj(Graphics g, Brush b)
{
    g.FillRectangle(new SolidBrush(Color.Green), x[0], y[0], segment, segment);
    for (int i = 1; i < segmenty; i++)
    {
        g.FillRectangle(b, x[i], y[i], segment, segment);
    }
}
```



Zadania

Pora na samodzielną pracę. **Rozwiązanie przedstawionych zadań znajdziemy w rozdziale czwartym.**

Zadanie 1

Stwórzmy proste **wyścigi samochodowe**, choć w nieco zmienionej formule – gdzie sterowanie pojazdem ogranicza się do zmiany toru jazdy pojazdu. Tory należy zmieniać tak, by uniknąć kolizji z przeciwnikami. Pojazdy mogą przemieszczać się po jednym z czterech torów.

Jak zacząć? Podpowiedź

Grę należy zrealizować jako **Aplikacja Windows Forms**. Do stworzenia grafik w grze można wykorzystać panel – podobnie jak w przypadku gry Snake.

Zadanie 2

Należy stworzyć grę, w której znajduje się określona liczba kulek swobodnie poruszających się po ekranie (gracz wybiera samodzielnie liczbę kulek, określając w ten sposób poziom trudności gry).

Kulki mogą odbijać się od krawędzi okna gry. Jedna kulka przeznaczona jest dla gracza. Może on sterować nią z pomocą klawiatury – w taki sposób, aby uciekać przed pozostałymi kulkami.

Jak zacząć? Podpowiedź

Gra powinna być stworzona w zgodzie z paradygmatem obiektowym. Podstawą do gry powinna być klasa **Kulka**, której obiekty będą poruszać się po ekranie.

4 Rozwiązania zadań w C#

PROGRAM I PLIKI PROJEKTÓW OPISANE W TYM ROZDZIALE ZNAJDZIESZ **NA DVD I W KŚ+**

Jeśli potrzebujemy pomocy w rozwiązywaniu zadań z poprzedniego rozdziału – oto propozycje rozwiązań krok po kroku. Każdy powinien poradzić sobie z ich realizacją według wskazówek

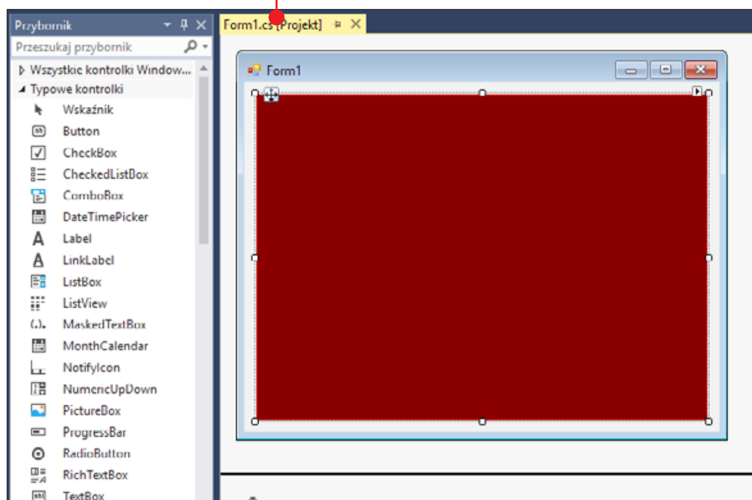
UWAGA!
Przedstawione rozwiązania to tylko propozycje. Istnieje wiele sposobów na realizację gier zaproponowanych w zadaniach. Ogromnym atutem umiejętności programowania jest to, że ten sam efekt można uzyskać na wiele sposobów.

Zadanie 1. Wyścigi samochodowe

Projektowanie okna
Pierwszy etap przy rozwiązywaniu tego zadania to zaprojektowanie okna gry.

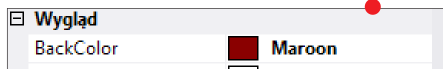
1 Z palety kontrolek należy wykorzystać dwie – **Panel** i **Timer**.

Pierwsza kontrolka posłuży do stworzenia pola gry – tak samo jak w projekcie z grą Snake.
Druga – również jak w projekcie z grą Snake – będzie odpowiadać za aktualizację grafik wyświetlanych na Panelu.



2 Nie musimy ustawiać konkretnych wymiarów Panelu. Na potrzeby gry będziemy sczytywać wymiary Panelu i dostosowywać wielkość i położenie grafik do jego aktualnych wymiarów.

3 W Panelu warto również zmienić wartość właściwości **BackColor** – aby ustawić początkowy kolor pola gry.



Klasa Przeciwnik

Nasza gra powinna być realizowana w zgodzie z paradygmatem obiektowym. Jeśli w grze pojawia się trzech takich samych przeciwników, którzy różnią się tylko szczegółami, należy stworzyć klasę, której obiektami będą poszczególni przeciwnicy.

1 Dlatego tworzymy nowy plik z klasą **Przeciwnik.cs**. Pisanie samej klasy rozpoczynamy od określenia jej pól. Każdy przeciwnik, a raczej samochód, reprezentowany będzie przez grafikę prostokąta. Grafika ta powinna mieć swoją lokalizację – współrzędne **x** i **y**, a także wymiary – **wysokość** i **szerokość**. Te cztery właściwości będą polami naszej klasy.

2 Współrzędna **x** nie może przyjąć dowolnej wartości mieszczącej się na polu gry. Współrzędna **x** może przyjmować tylko takie wartości, aby samochód znajdował się na środku jednego z czterech pasów ruchu. Wartość współrzędnej **x** jest zatem zależna od tego, na którym pasie ruchu znajduje się przeciwnik. Możemy dopisać do klasy jeszcze pole **pas** – które będzie przyjmować wartość od **0** do **3**.

```
class Przeciwnik
{
    public int pas;
    public int x;
    public int y;
    public int szerokosc;
    public int wysokosc;
```

Dlaczego nie od 1 do 4? Ze względu na to, że na podstawie numeru pasa wyliczana będzie wartość współrzędnej **x**, a takie ponumerowanie pasów ułatwi nam późniejsze obliczenia.

3 Kiedy pola klasy są zadeklarowane, należy nadać im wartości. Zrobimy to oczywiście poprzez konstruktor. Dla przeciwnika konstruktor będzie pobierał trzy parametry. Wymiary pola gry są nam potrzebne do obliczenia wymiarów przeciwnika – jego wysokość i szerokość powinny stanowić **10%** wysokości i szerokości pola gry, które będą przekazane do konstruktora. Trzeci parametr to wartość współrzędnej **y**. Tworząc kolejnych przeciwników, będziemy mogli dawać im takie **y**, aby przeciwnicy nie nachodzili na siebie.

```
public Przeciwnik(int szerokosc_okna, int wysokosc_okna, int y)
{
    wysokosc = wysokosc_okna / 10;
    szerokosc = szerokosc_okna / 10;
    this.y = y;
    Random r = new Random();
    pas = r.Next(0, 4);
    int szerokosc_pasa = szerokosc_okna / 4;
    x = (pas * (szerokosc_pasa))+(szerokosc_pasa/2)-(szerokosc/2);
}
```

4 Wartość pola **pas** zostanie wylosowana – dlatego tworzymy generator liczb pseudolosowych (obiekt klasy **Random**). Za pomocą generatora losujemy liczbę z zakresu 0–4, przy czym 0 jest najmniejszą możliwą liczbą, a 4 to górna granica losowania, od której największa wylosowana liczba musi być mniejsza. Dlatego pas dostanie wartość 0, 1, 2 lub 3.

5 Dalej możemy przejść do wyliczenia wartości współrzędnej **x**. Aby było nam łatwiej, stworzymy zmienną pomocniczą, która będzie przechowywać szerokość jednego pasa ruchu. Ponieważ mamy cztery pasy, szerokość jednego jest szerokością pola gry podzieloną przez 4.

6 Pole **x** musi dostać taką wartość, aby gdy będziemy rysować prostokąt symbolizujący samochód, mógł się on znaleźć na środku

```
public Przeciwnik(int szerokosc_okna, int wysokosc_okna, int y)
{
    wysokosc = wysokosc_okna / 10;
    szerokosc = szerokosc_okna / 10;
    this.y = y;
    Random r = new Random();
    pas = r.Next(0, 4);
    int szerokosc_pasa = szerokosc_okna / 4;
    x = (pas * (szerokosc_pasa))+(szerokosc_pasa/2)-(szerokosc/2);
}
```

pasa ruchu. Współrzędna **x** podczas rysowania mówi o lewej krawędzi prostokąta. Współrzędną **x** lewej krawędzi pasa ruchu będzie wynik mnożenia pola **pas** przez szerokość pasa: ***x = (pas * (szerokosc_pasa))***. My jednak musimy odsunąć się od krawędzi – w stronę środka. Aby **x** był równy środkowi pasa, należy dodać jeszcze połowę szerokości pasa: ***x = (pas * (szerokosc_pasa))+(szerokosc_pasa/2)***. Nie możemy jednak rysować przeciwnika w tym miejscu – teraz lewa krawędź prostokąta zaczynałaby się na środku pasa ruchu. Aby środek prostokąta pokrywał się ze środkiem pasa ruchu, od wyniku poprzednich obliczeń trzeba odjąć jeszcze połowę szerokości prostokąta: ***x = (pas * (szerokosc_pasa))+(szerokosc_pasa/2)-(szerokosc/2)***.

7 Projekt ten ma bardzo dużo analogii z grą Snake. Bardzo podobnie wygląda na przykład metoda klasy **Przeciwnik** odpowiadająca za rysowanie. Tu również wykorzystamy obiekt graficzny do wywołania funkcji **DrawRectangle**, dla której parametrami będą wartości pól obiektu,

```
public void rysuj(Graphics g, Brush b)
{
    g.FillRectangle(b, x, y, szerokosc, wysokosc);
}
```

```
using System;
using System.Collection
using System.Linq;
using System.Text;
using System.Threading.
using System.Drawing;
```

a także pędzel, utworzony bezpośrednio przy wywołaniu metody.

8 Aby rysowanie działało poprawnie, u góry klasy trzeba napisać jeszcze **using System.Drawing;**. Bez tego nie uda nam się w ten sposób skorzystać z obiektów klas **Graphics** i **Brush**.

9 Każdy przeciwnik powinien przesuwać się do dołu pola gry. Dlatego tworzymy metodę ***przesuń***, która wywołana dla obiektu zmieni wartość jego współrzędnej **y**, powiększając ją o **10**. Metoda ta będzie wywoływana poprzez **zdarzenie Timera**.

```
public void przesun()
{
    y = y + 10;
}
```

10 Nie możemy jednak zwiększać współrzędnej **y** w nieskończoność. Należy sprawdzić, czy nie osiągnęła ona wartości takiej, że przeciwnik będzie znajdował się poza polem gry.
Pamiętamy z konstruktora, że wysokość przeciwnika to 1/10 wysokości pola gry, dlatego 10 wysokości przeciwnika stanowi wysokość pola gry.
Jeśli **y** będzie większe niż wysokość pola gry, należy przenieść przeciwnika na górę, ustawiając jego **y** na **0**.

```
public void przesun()
{
    y = y + 10;
    if (y > wysokosc * 10)
    {
        y = 0;
        Random r = new Random();
        pas = r.Next(0, 4);
        int szerokosc_pasa = (szerokosc*10) / 4;
        x = (pas * (szerokosc_pasa)) + (szerokosc_pasa / 2) - (szerokosc / 2);
    }
}
```

11 Oprócz przeniesienia przeciwnika do góry należy wyznaczyć dla niego nowy pas ruchu (wylosować go), a także wyliczyć wartość współrzędnej **x**. Robimy to identycznie jak w konstruktorze (z tą różnicą, że za szerokość pola gry przyjmujemy dziesięciokrotność szerokości przeciwnika).

12 Do zakończenia definicji klasy **Przeciwnik** pozostaje nam już tylko stworzenie jednej metody – metody typu boolowskiego, która będzie zwracać prawdę, gdy przeciwnik jest w kolizji z graczem, lub fałsz, gdy nie jest. Dlatego metoda ta pobiera w parametrze współrzędną **x** gracza. Jeśli współrzędne **x** przeciwnika i gracza są takie same, należy sprawdzić, czy dolna krawędź

```
public bool kolizja(int x_auto)
{
    if (x_auto == x)
    {
        if (y + wysokosc > wysokosc * 9)
        {
            return true;
        }
    }
    return false;
}
```

przeciwnika (znajdująca się o wysokość pikseli poniżej współrzędnej **y** przeciwnika) nachodzi na górną krawędź gracza (znajdującą się o wysokość pikseli ponad dolną krawędzią pola gry). Gdy i ten warunek jest spełniony – zachodzi kolizja pomiędzy postaciami. Należy wtedy zwrócić wartość **true**. Gdy wartość **true** nie zostanie zwrócona w danym wywołaniu metody ***kolizja***, należy zwrócić wartość **false**.

Klasa Samochod

Gracz również powinien mieć do dyspozycji swój samochód. Nie może on być jednak obiektem klasy **Przeciwnik**. Aby utworzyć obiekt, którym będzie sterował gracz, stworzymy nową klasę **Samochod**.

1 Nową klasę możemy umieścić w nowym pliku z klasą – **Samochod.cs**.

2 Klasa **Samochod** będzie podobna do **Przeciwnika**, nie obędzie się jednak bez znaczących różnic. **Przeciwnik** miał pola **pas**, **x**, **y**, **szerokosc** i **wysokosc** – samochód gracza również może poruszać się po czterech pasach i zmieniać swoje **x**. Będzie miał także wymiary określone na podstawie wysokości i szerokości pola gry. Nie musimy jednak tworzyć w nim pola **y**, gdyż współrzędną tę bez problemu obliczymy na podstawie wysokości samochodu gracza. Zatem klasa **Samochod** powinna mieć pola **pas**, **x**, **wysokosc** i **szerokosc**.

```
class Samochod
{
    public int pas;
    public int x;
    public int szerokosc;
    public int wysokosc;
```

3 Konstruktor obiektów klasy **Samochod** również będzie podobny do **Przeciwnika**. Główną różnicą jest to, że nie przyjmuje on w parametrze wartości współrzędnej **y**, w której ma znaleźć się górna krawędź prostokąta symbolizującego samochód gracza. Przyjmuje za to wymiary pola gry. Należy je wykorzystać do obliczenia wysokości i szerokości samochodu gracza, które, podobnie jak w przypadku przeciwników, stanowić będą 1/10 wymiarów pola gry. Podobnie jak w przypadku samochodów przeciwnika generowana jest losowa liczba mówiąca o pasie, na którym znajdować się będzie samochód gracza. W identyczny sposób wyliczamy też współrzędną **x** samochodu – na podstawie numeru pasa ruchu.

```
public Samochod(int szerokosc_okna, int wysokosc_okna)
{
    wysokosc = wysokosc_okna / 10;
    szerokosc = szerokosc_okna / 10;
    Random r = new Random();
    pas = r.Next(0, 4);
    int szerokosc_pasa = szerokosc_okna / 4;
    x = (pas * (szerokosc_pasa)) + (szerokosc_pasa / 2) - (szerokosc / 2);
}
```

```
public void rysuj(Graphics g, Brush b)
{
    g.FillRectangle(b, x, wysokosc*9, szerokosc, wysokosc);
}
```

4 Klasa **Samochod** też powinna mieć metodę odpowiedzialną za rysowanie. Będzie ona bardzo podobna do metody ***rysuj*** z klasy **Przeciwnik**. Jedyną różnicą będzie to, że zamiast pola **y** do wyznaczenia górnej krawędzi rysowanego prostokąta wykorzystamy wynik mnożenia wysokości prostokąta przez **9** – w efekcie prostokąt zostanie stworzony w takim miejscu, że jego dolna krawędź będzie pokrywała się z dolną krawędzią pola gry.

5 Ostatnią metodą klasy **Samochod** będzie ***przesun***, a to oznacza, że klasa ta nie będzie mieć metody ***kolizja*** do sprawdzania zderzenia z innym samochodem. Jednak definicja i działanie metody ***przesun*** będą zupełnie inne niż w klasie **Przeciwnik**. Tym razem przesuwanie będzie odbywało się po osi **X** – przesuwanie to tak naprawdę zmiana pasa ruchu (**y** dla tego samochodu będzie stałe w trakcie trwania gry).
Metoda ta będzie przyjmowała parametr w typie **string** – wywołując metodę, będziemy w nim wpisywać **prawa** lub **lewa**, określając stronę, w którą ma przesunąć się samochód. Sprawdzamy wartość wpisaną w parametr i jeśli jest ona równa **prawa** – to znaczy, że chcemy przesunąć samochód w prawą stronę. Możemy to zrobić, tylko jeśli nie znajdujemy się na skrajnym prawym pasie. To oznacza, że numer pasa musi być mniejszy niż **3**. Jeśli numer pasa jest mniejszy niż **3**, zwiększamy numer pasa. Kiedy w parametrze nie pojawi się słowo **prawa**, możemy założyć, że chcemy przesunąć się w lewą stronę. Jeśli chcemy przesunąć się w lewą stronę – możemy zrobić to, tylko gdy numer pasa ruchu jest większy niż **0** (czyli nie znajdujemy się na skrajnym lewym pasie ruchu).

6 Po zmianach wartości numeru pasa ruchu należy wyliczyć nową wartość współrzędnej **x** dla danego pasa ruchu. Sposób obliczeń jest taki sam jak w poprzednich przypadkach.

Tworzymy obiekty klas i programujemy okno programu

Czas przenieść się do klasy **Form1** i w niej kontynuować pisanie, między innymi tworząc obiekty napisanych przez nas wcześniej klas.

1 Rozpoczynamy od uruchomienia **Timera**. W konstruktorze klasy **Form1**

```
public void przesun(string strona)
{
    if (strona == "prawa")
    {
        if(pas < 3)
        {
            pas = pas + 1;
        }
    }
    else
    {
        if (pas > 0)
        {
            pas = pas - 1;
        }
    }
    int szerokosc_pasa = (szerokosc*10) / 4;
    x = (pas * (szerokosc_pasa)) + (szerokosc_pasa / 2) - (szerokosc / 2);
}
```

możemy wpisać przypisanie wartości **true** do właściwości ***Enabled*** w **Timerze**. To już po starcie programu uruchomi **Timer**, który swoje zdarzenie w tej grze powinien zgłaszać co 100 milisekund – co ustawiamy poprzez właściwość **Interval** Timera.

```
public Form1()
{
    InitializeComponent();
    timer1.Enabled = true;
    timer1.Interval = 100;
}
```

2 **Timer** zgłasza już swoje zdarzenie **Tick** – musimy teraz umieścić w nim kod. **Timer** będzie działać inaczej, gdy gra będzie włączona, a inaczej, zanim ją włączymy. Aby oddzielić te dwa stany, deklarujemy pole klasy **Form1** – typu boolowskiego – **czy_gra_aktywna**. Od razu przypisujemy temu polu wartość **false**, ponieważ na początku gra nie jest aktywna i należy ją aktywować.

```
bool czy_gra_aktywna = false;

public Form1()
{
    InitializeComponent();
    timer1.Enabled = true;
    timer1.Interval = 100;
}
```

3 Teraz możemy przejść do zdarzenia wywoływanego przez **Timer**. Aby wygenerować metodę przypisaną do tego zdarzenia, należy przejść do okna modelowania programu i w nim dwukrotnie kliknąć na dodany do projektu **Timer**. W wygenerowanej przez to procedurze rozpoczynamy pisanie od utworzenia instrukcji warunkowej **if**. Będzie ona sprawdzać wartość pola **czy_gra_aktywna**, aby określić, co należy wyrysować w polu gry. Na razie nie mamy jeszcze obiektów, które należy narysować, jeśli gra jest aktywna, dlatego najpierw zajmiemy się napisaniem tego, co należy zrobić, gdy gra nie jest aktywna (sekcja **else** instrukcji warunkowej). Tu pojawia się kolejna analogia do projektu z grą Snake. Tam, gdy gra nie była aktywna – wypisywaliśmy na Panelu informację o tym, jak ją włączyć. Tym razem zrobimy podobnie, jednak inny będzie sposób uruchomienia gry. W tym projekcie nie mamy menu, dlatego nie możemy poinformować gracza o tym, żeby kliknął na przycisk **Start**. Możemy mu jednak napisać, aby kliknął na **Panel**. W jaki sposób to wypisać? Tak samo, jak zrobiliśmy to w przypadku gry Snake. Należy jednak pamiętać o tym, aby dobrać odpowiednie współrzędne dla lokalizacji napisu, aby pasował on do naszego Panelu. (jeżeli zrobimy Panel o innych wymiarach niż w opisywanym przykładzie, podana lokalizacja napisu może nie być odpowiednia).

4 Abyśmy mogli rysować poszczególne samochody (czy to gracza, czy przeciwników) – musimy mieć obiekty. Deklarujemy (jako pola klasy **Form1**) trzech przeciwni-

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {

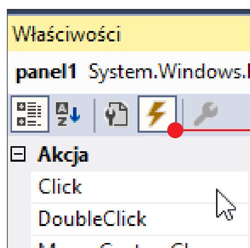
    }
    else
    {
        FontFamily fontFamily1 = new FontFamily("Arial");
        Font f = new Font(fontFamily1, 15);
        Brush b = new SolidBrush(Color.Aqua);
        panel1.CreateGraphics().DrawString("Kliknij, aby rozpocząć", f, b, 120, 135);
    }
}
```

```
public partial class Form1 : Form
{
    Przeciwnik p1;
    Przeciwnik p2;
    Przeciwnik p3;
    Samochod gracz;
    bool czy_gra_aktywna = false;

    public Form1()
    {
        InitializeComponent();
        timer1.Enabled = true;
        timer1.Interval = 100;
    }
```

ków – jako obiekty klasy **Przeciwnik**, i jeden obiekt klasy **Samochod** – jako auto gracza.

5 Polom tym należy nadać wartości przez wywołanie konstruktorów – a te należy wywołać przez kliknięcie na **Panel**. Przechodzimy zatem do okna modelowania i w nim do sekcji **Właściwości**, w której przełączamy się do zdarzeń przyciskiem z ikoną błyskawicy. Odszukujemy zdarzenie **Click** dla Panelu i klikamy na nie dwukrotnie.



6 Wygenerowana zostanie metoda odpowiadająca za kliknięcie na **Panel**. W niej dla pól klasy **Form1** wywołujemy konstruktory napisanych przez nas klas. W ich parametrach jako szerokość i wysokość okna gry podajemy szerokość i wysokość Panelu. Dla przeciwników podajemy również współrzędną **y** – należy użyć takich wartości, aby dla każdego przeciwnika **y** było inne. Dodatkowo **y** przeciwników musi różnić się co najmniej o **1/10** wysokości Panelu. To zagwarantuje nam, że gdy przeciwnicy pojawią się na jednym pasie ruchu, nie będą na siebie nachodzić.

7 Oprócz tego w metodzie opowiadającej za zdarzenie kliknięcia na **Panel** powinniśmy nadać wartość **true** polu **czy_gra_aktywna**, gdyż kliknięcie na **Panel** uruchamia grę.

8 Możemy powrócić teraz do zdarzenia **Timera** i w nim, gdy gra jest aktywna, napisać wywołanie metod ***rysuj*** **A** dla czterech obiektów. Pamiętajmy, że działamy jakby w pętli, dlatego przed każdym kolejnym wyrysowaniem tych obiektów należy „pozbyć się" poprzednich wyrysowań – stąd przed rysowaniem używamy metody ***Clear*** **B** dla obiektu graficznego z Panelu.

```
private void panel1_Click(object sender, EventArgs e)
{
    p1 = new Przeciwnik(panel1.Width, panel1.Height, panel1.Height / 10);
    p2 = new Przeciwnik(panel1.Width, panel1.Height, (panel1.Height / 10) * 3);
    p3 = new Przeciwnik(panel1.Width, panel1.Height, (panel1.Height / 10) * 6);
    gracz = new Samochod(panel1.Width, panel1.Height);
    czy_gra_aktywna = true;

}
```

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {                                          B
        panel1.CreateGraphics().Clear(Color.Black);

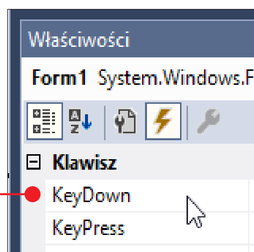
        p1.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        p2.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
  A     p3.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        gracz.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.Green));
```

9 Aby kolejne rysowania miały sens, należy obiekty wyrysowywać w innych miejscach. W zdarzeniu **Timera** wywołujemy więc też metodę ***przesun*** dla każdego obiektu przeciwnika.

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
        panel1.CreateGraphics().Clear(Color.Black);

        p1.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        p2.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        p3.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        gracz.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.Green));
        p1.przesun();
        p2.przesun();
        p3.przesun();
```

10 Jeśli przesuwamy przeciwników – przesuwać powinien się też gracz. Dla niego przesunięcie będziemy wywoływać, gdy osoba grająca w naszą grę naciśnie klawisz strzałki w lewą lub prawą stronę. Aby reagować na zdarzenia wciśnięcia klawisza na klawiaturze, musimy przenieść się do modelowania i w części **Właściwości** dla okna gry przenieść się do zdarzeń, wśród których znajduje się zdarzenie **KeyDown**, na które należy dwukrotnie kliknąć, aby wygenerować odpowiednią metodę.



11 W treści wygenerowanej metody poprzez instrukcje warunkowe **if** sprawdzamy, jaki jest kod naciśniętego klawisza i jeśli odpowiada on strzałce w prawą stronę i strzałce w lewą stronę – wywołujemy metodę **przesun** dla gracza. W jej parametrze należy wpisać stronę, w którą chcemy przesunąć prostokąt zależnie od tego, która ze strzałek została właśnie naciśnięta.

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Right) gracz.przesun("prawa");
    if (e.KeyCode == Keys.Left) gracz.przesun("lewa");
}
```

12 Po przesunięciu zarówno gracza, jak i przeciwników należałoby sprawdzić, czy nie wchodzą oni ze sobą w kolizje. Dlatego w zdarzeniu **Timera** dopisujemy instrukcję **if**, której warunkiem do spełnienia powinno być to, że w pierwszym, drugim lub trzecim przeciwniku wywołanie metody ***kolizja*** zwróci prawdę. Metoda ta musi być zawsze wywoływana z parametrem, którym jest wartość współrzędnej **x** gracza.

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
        panel1.CreateGraphics().Clear(Color.Black);

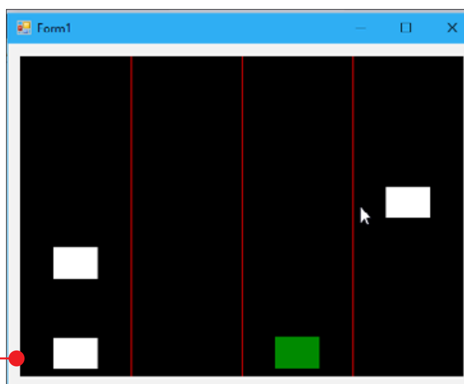
        p1.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        p2.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        p3.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        gracz.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.Green));
        p1.przesun();
        p2.przesun();
        p3.przesun();
        if (p1.kolizja(gracz.x) || p2.kolizja(gracz.x) || p3.kolizja(gracz.x))
        {
            czy_gra_aktywna = false;
        }
```

13 Kolejnym, nieobowiązkowym, jednak poprawiającym estetykę gry ele-

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {
        panel1.CreateGraphics().Clear(Color.Black);

        p1.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        p2.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        p3.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.White));
        gracz.rysuj(panel1.CreateGraphics(), new SolidBrush(Color.Green));
        p1.przesun();
        p2.przesun();
        p3.przesun();
        if (p1.kolizja(gracz.x) || p2.kolizja(gracz.x) || p3.kolizja(gracz.x))
        {
            czy_gra_aktywna = false;
        }
        panel1.CreateGraphics().DrawLine(new Pen(new SolidBrush(Color.Red)), new Point(panel1.Width / 4, 0), new Point(panel1.Wi
        panel1.CreateGraphics().DrawLine(new Pen(new SolidBrush(Color.Red)), new Point(panel1.Width / 2, 0), new Point(panel1.Wi
        panel1.CreateGraphics().DrawLine(new Pen(new SolidBrush(Color.Red)), new Point((panel1.Width / 4) * 3, 0), new Point((pa
```

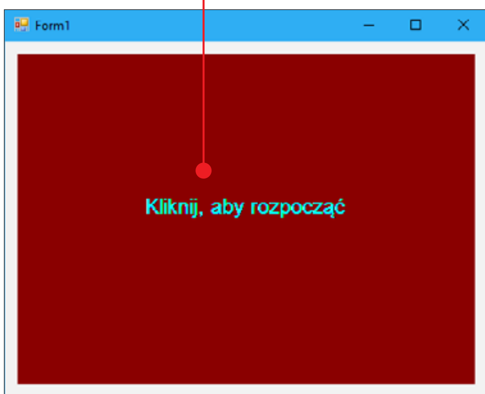
mentem kodu może być trzykrotne wywołanie polecenia **DrawLine** ● – aby narysować linie oddzielające pasy ruchu.



Testowanie

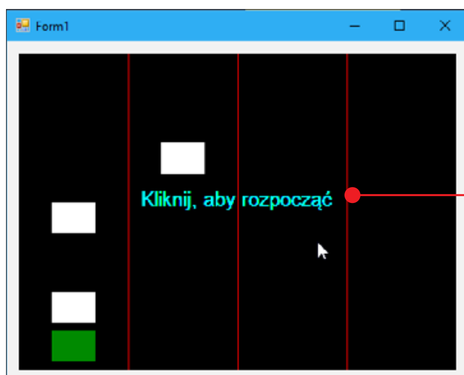
W ten sposób napisaliśmy już cały kod gry. Dobrze jest sprawdzać jego działanie sukcesywnie, po dopisywaniu kolejnych kroków, jednak najważniejsze jest sprawdzenie poprawności działania całego kodu.

1 Po uruchomieniu gry powinien pojawić się Panel w kolorze tła, jaki ustawiliśmy jako **BackColor** Panelu. Na nim powinien widnieć napis ●.



2 Po kliknięciu na **Panel** gra powinna się uruchomić ●. Możemy już grać, przesuwając samochód tak, aby unikać przeciwników.

3 Kiedy wystąpi zderzenie z przeciwnikiem, gra powinna się zatrzymać i pojawić się powinien napis ●, który był widoczny także bezpośrednio po uruchomieniu gry. Ponowne kliknięcie na **Panel** restartuje grę.



DOKOŃCZ SAMODZIELNIE*

Zaprezentowane rozwiązanie zadania dostarcza działającą grę. Można ją jednak znacznie bardziej dopracować. Oto kilka pomysłów:
1. Dodaj system zliczania i wyświetlania punktów.
2. Zastąp napis Form1 na pasku tytułowym nazwą gry.
3. Zastąp standardową grafikę na pasku tytułowym własną ikoną gry.

***Podpowiedź: 1.** Tworzymy pole **punkty** i zwiększamy jego wartość, gdy samochód przeciwnika opuszcza pole gry. Dodajemy też etykietę, na której będziemy wyświetlać wartość pola punkty.
2. i **3.** W oknie modelowania gry należy zmienić wartości we właściwościach formatki.

Zadanie 2. Kulki

Zadanie polega na stworzeniu gry, w której podana przez gracza liczba kulek będzie poruszać się po oknie gry w losowych kierunkach. W grze powinna występować też postać sterowana przez gracza za pomocą strzałek. Zderzenie postaci z jedną z kulek spowoduje zakończenie gry.

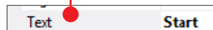
Modelowanie okna gry

Pierwsze zadanie bazowało na znanej nam grze Snake. Drugie zadanie również jest do niej podobne. W tym projekcie, jak w dwóch poprzednich – będziemy wykorzystywać Panel, na którym wyrysowane zostaną grafiki kulek.

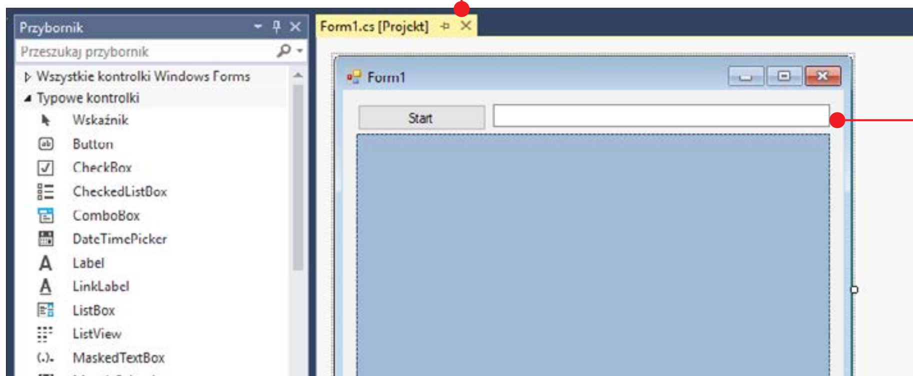
1 To oznacza, że z Przybornika przeciągamy do okna modelowania właśnie Panel. Możemy powiększyć wymiary okna, a następnie rozszerzyć dodany przez nas Panel, aby wypełnił okno.

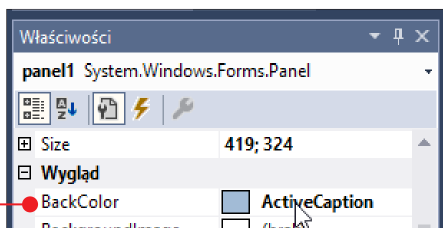
2 Ponieważ gracz powinien sam wpisać liczbę kulek, dodajemy **Textbox**, czyli pole tekstowe, w którym gracz będzie mógł pisać.

3 W modelu okna umieszczamy też przycisk, czyli **Button**, na który kliknięcie wystartuje grę. Dlatego na przycisku powinien znaleźć się napis **Start**. Aby tak było, powinniśmy zmienić właściwość **Text** w sekcji **Właściwości** przycisku.



4 Panel powinien być wyraźniej zaznaczony – nie powinien być w kolorze tła okna, ponieważ wtedy nie widać jego krawę-





dzi. Zmieńmy kolor Panelu poprzez zmianę wartości jego właściwości **BackColor**.

5 Podobnie jak w przypadku poprzednich projektów w Panelu będą wyrysowywane obiekty, które mogą zmieniać swoje położenie. Zmiana położenia i ponowne wyrysowanie obiektów będzie musiało wykonywać się w swego rodzaju pętli. Polega ona na cyklicznym wywoływaniu zdarzenia **Timera**. Niezbędne jest zatem dodanie go do projektu.

Klasa Kulka

Po zamodelowaniu okna gry możemy przejść do pisania, które rozpoczniemy od stworzenia nowej klasy. Aby zapisać klasę **Kulka** możemy utworzyć nowy plik z klasą – **Kulka.cs**.

1 Tworzenie naszej klasy zaczynamy od opisania pól, jakie powinny się w niej znaleźć. Kulka będzie czymś, co należy narysować na Panelu. Aby ją narysować, należy znać jej położenie. Kulka musi mieć zatem pola **x** i **y**, które będą przechowywać wartości jej współrzędnych.

2 Wszystkie kulki muszą mieć też określone wymiary. Tym razem nie będziemy mówić o wysokości i szerokości kulki, ale o jej **średnicy**, która podobnie jak współrzędne **x** i **y** zapisana zostanie jako pole w typie **Integer**.

3 Każda kulka powinna się poruszać. Ruch kulki to zmiana współrzędnej **x**, **y** lub ich obu. Zmieniając **x** i **y**, będziemy dodawać do nich pewne wartości. Jeśli do współrzędnej **x** dodamy liczbę o wartości dodatniej, **x** wzrośnie, a obiekt przesunie się w prawą stronę. Jeśli do **x** dodamy liczbę o wartości ujemnej – **x** zmaleje, a obiekt przesunie się w stronę lewą. Jakie liczby będziemy zatem dodawać do **x** i **y**? Te liczby to **dx** (dodawany do **x**) i **dy** (dodawane do **y**). Ich wartości będą dodatnie lub ujemne – zależnie od tego, w którą stronę będzie miał poruszać się obiekt. Jeśli obiekt porusza się w prawą stronę i dotknie krawędzi pola gry, zmienimy wartość **dx** na ujemną, co zmieni kierunek ruchu obiektu. W ten sposób zaprogramujemy odbijanie kulek od ścian pola gry. Deklarujemy zatem **dx** i **dy** jako pola klasy **Kulka**.

```
class Kulka
{
    public int x;
    public int y;
    public int srednica;
    public int dx;
    public int dy;
    public Color kolor;
```

4 Ostatnim polem klasy **Kulka** będzie **kolor** – przyjmie ono wartość obiektu klasy **Color**, który posłuży nam do wywołania metody rysującej, aby nadać odpowiedni kolor kulce. Abyśmy mogli stworzyć pole takiego typu, w naszym skrypcie powinna pojawić się linia ***using System.Drawing;***.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;
```

5 Po zadeklarowaniu wszystkich pól klasy **Kulka** możemy przejść do utworzenia konstruktora obiektu tej klasy. Konstruktor ten przyjmie trzy parametry. Dwa z nich odpowiadają wymiarom pola gry – po jednym dla wysokości i szerokości. Trzeci parametr to obiekt typu **Random**, który posłuży

```
public Kulka(int szer, int wys, Random r)
{
    srednica = r.Next(4, 15);
```

nam do otrzymania losowych wartości. Będziemy tworzyć wiele kulek, dlatego chcemy, aby jak najbardziej różniły się od siebie.

Pierwszą różnicą pomiędzy kulkami będzie ich rozmiar. Pole **srednica** powinno do-

```
public Kulka(int szer, int wys, Random r)
{
    srednica = r.Next(4, 15);
```

stać losową wartość, w naszym przykładzie z przedziału 4–15, ale można użyć innego przedziału, na przykład 10–20, aby uzyskać większe kulki.

6 Kulki powinny znajdować się w różnych miejscach. Dlatego wartość pola mówiącego o współrzędnej **x** kulki powinna być losowa. Pole to opisuje lokalizację lewego boku kulki. Musi ono zatem dostać taką wartość losową, aby cała kulka mieściła się na polu gry. Najmniejsze możliwe **x** dla kul-

```
public Kulka(int szer, int wys, Random r)
{
    srednica = r.Next(4, 15);
    x = r.Next(0, szer - srednica);
```

ki to **0** – wpisujemy **0** jako dolną granicę losowania. Za górną granicę losowania nie może posłużyć nam sama szerokość pola gry. Jeśli kulka dostanie **x** takie jak szerokość pola gry, to w tym miejscu znajdzie się jej lewy brzeg, co oznacza, że cała kulka znajdzie się poza polem gry. Aby kulka zmieściła się na polu gry, wartość jej współrzędnej **x** nie może być większa niż szerokość pola gry pomniejszona o szerokość kulki (w tym wypadku o jej średnicę).

7 Identyczna sytuacja, jak w przypadku współrzędnej **x**, występuje przy współrzędnej **y**. Ona również powinna dostać wartość losową. Najmniejsze możliwe **y** dla kulki to **0**. Współrzędna ta musi być jednak

```
public Kulka(int szer, int wys, Random r)
{
    srednica = r.Next(4, 15);
    x = r.Next(0, szer - srednica);
    y = r.Next(0, wys- srednica);
```

mniejsza niż wysokość okna pomniejszona o wysokość (czyli również średnicę) kulki.

8 Losowe wartości powinny dostać też pola **dx** i **dy** mówiące o kierunku ruchu kulki. Tym razem możemy dać zakres losowania taki, aby możliwe było uzyskanie zarówno dodatniej, jak i ujemnej liczby. W naszym przykładzie mamy zakres od **-2** do **2**. W nim możliwe do uzyskania wartości to **-2**,

```
public Kulka(int szer, int wys, Random r)
{
    srednica = r.Next(4, 15);
    x = r.Next(0, szer - srednica);
    y = r.Next(0, wys- srednica);
    dx = r.Next(-2, 2);
    dy = r.Next(-2, 2);
```

-1, **0** i **1**. Każdy może użyć jednak zupełnie innego zakresu – choćby od **-2** do **3**. Pamiętajmy jednak, że wartość bezwzględna **dx** i **dy** mówi o prędkości kulki. Nie jest ważne tylko to, czy liczby te są dodatnie czy ujemne, ale też jak duże są to liczby. Kulka z **dx** i **dy** na poziomie **-5** lub **5** będzie pięć razy szybsza niż kulka mająca w tych polach wartości **-1** lub **1**.

```
public int x;
public int y;
public int srednica;
public int dx;
public int dy;
public Color kolor;
public Kulka(int szer, int wys, Random r)
{
    srednica = r.Next(4, 15);
    x = r.Next(0, szer - srednica);
    y = r.Next(0, wys- srednica);
    dx = r.Next(-2, 2);
    dy = r.Next(-2, 2);
    kolor = Color.FromArgb(r.Next(256), r.Next(256), r.Next(256));
}
```

9 Ostatnie pole również dostanie losową **wartość**. Jak uzyskać losowy kolor? Tu z pomocą przychodzi nam system RGB. Każdy kolor można zapisać jako kombinację czerwonego, zielonego i niebieskiego (RGB to skrót od Red Green Blue). Poprzez funkcję ***FromArgb*** możemy uzyskać kolor w systemie RGB. Każdy parametr odpowiadający

za jeden z trzech bazowych kolorów może przyjąć wartość od **0** do **255**. Stąd liczba **256** jako górna granica losowania (gdy podamy w parametrze metody **Next** tylko jedną liczbę, będzie ona górną granicą losowania, a dolna granica to domyślnie **0**).

10 Po nadaniu wszystkim polom klasy początkowych wartości możemy przejść do definicji metody rysującej kulkę. Tym razem jest ona nieco inna niż metody rysujące w poprzednich projektach. To dlatego, że klasa **Kulka** ma pole **kolor**. W poprzednich projektach metody rysujące pobierały dwa parametry – obiekt graficzny i pędzel. Tym razem potrzebujemy tylko obiektu graficznego. Pędzel utworzymy bezpośrednio w parametrze funkcji **FillEllipse**,

```
public void rysuj(Graphics g)
{
    g.FillEllipse(new SolidBrush(kolor), x, y, srednica, srednica);
}
```

używając konstruktora obiektu **SolidBrush**, który za parametr pobierze wartość pola **kolor**. Pozostałe parametry funkcji **FillEllipse** to współrzędne **x** i **y** kulki, a także jej wymiary. Musimy podać szerokość i wysokość kulki – których rolę w naszej klasie pełni pole **srednica**.

11 Podobnie jak w poprzednich projektach klasa **Kulka** będzie miała metodę **przesun**, która tym razem polega na zmianie współrzędnych **x** i **y** o wartości odpowiednio pól **dx** i **dy**.

```
public void przesun()
{
    x = x + dx;
    y = y + dy;
}
```

12 Nie możemy jednak przesuwać kulki bez ograniczeń, ponieważ w pewnym momencie wyjdzie ona poza granice pola gry. Lewa granica pola gry to **x = 0**, prawa granica pola gry to **y** równe jego szerokości. Analogicznie jest ze współrzędną **y** i wysokością pola gry, gdzie **y = 0** to górna granica, a dolna to **y** równe wysokości. Aby móc uwzględnić wymiary pola gry w ruchu kulki, należy przekazać te wymiary w parametrach metody **przesun**.

```
public void przesun(int szer, int wys)
{
    x = x + dx;
    y = y + dy;
}
```

13 Teraz musimy skorzystać z pozyskanych przez parametr wartości. Przed przesunięciem kulki powinniśmy napisać dwie instrukcje warunkowe **if**. Jedna będzie dotyczyć osi X i sprawdzać to, czy kulka dotyka swoimi brzegami lewej lub prawej krawędzi okna. Dwie pionowe kreski, które oddzielają dwa warunki w **if**, oznaczają **Or** (albo) – wystarczy, że jeden z tych dwóch warunków jest spełniony, a wykona się treść instrukcji. A jaka będzie treść instrukcji? To zmiana kierunku kulki na osi X, czyli przemnożenie **dx** przez **-1**. To sprawia, że jeśli **dx** było ujemne, stanie się dodatnie, a jeśli było dodatnie, stanie się ujemne. Identyczny manewr powtarzamy w drugiej instrukcji warunkowej, gdzie badając współrzędną **y**, sprawdzamy, czy kulka dotyka górnej lub dolnej krawędzi pola gry. Jeśli tak jest, zmieniamy kierunek na osi **y** poprzez przemnożenie wartości pola **dy** przez **-1**.

```
public void przesun(int szer, int wys)
{
    if (x < 0 || x + srednica > szer)
    {
        dx = dx * -1;
    }
    if (y < 0 || y + srednica > wys)
    {
        dy = dy * -1;
    }
    x = x + dx;
    y = y + dy;
}
```

14 Do zakończenia definicji klasy **Kulka** pozostało nam już tylko napisanie metody, która będzie zwracać informację o tym,

```
public bool czy_kolizja(int x_gracz, int y_gracz, int szer_gracz, int wys_gracz)
{
    if (x+srednica>x_gracz && x < x_gracz + szer_gracz)
    {

    }
}
```

czy kulka znajduje się w kolizji z obiektem gracza. Deklarując metodę ***czy_kolizja***, określamy typ jako **bool** – zwracana informacja może być tylko prawdą lub fałszem. Aby jednak określić, czy kolizja występuje, musimy znać lokalizację i wymiary obiektu gracza. Te wartości przekazujemy poprzez parametry metody.

15 Treścią metody ***czy_kolizja*** będą dwie instrukcje warunkowe **if** – jedna wewnątrz drugiej. W pierwszej sprawdzimy, czy kulka pokrywa się z obiektem gracza na osi X, a w drugiej – czy kulka pokrywa się z obiektem gracza na osi Y. Kiedy warunek w wewnętrznym **if** jest spełniony, należy zwrócić **true**. To oznacza, że kolizja występuje. Nazwa metody wciąż jest jednak podkreślona, co świadczy o tym, że mamy błąd w skrypcie. Polega on na tym, że jeśli ify nie są spełnione – metoda nic nie zwróci. Metoda musi coś zwracać. Dopisujemy zatem pod instrukcjami warunkowymi **return false;** – co oznacza, że jeśli wcześniej metoda nie zwróciła prawdy, a kod wykonał się do tego miejsca (po słowie **return** metoda przestaje się dalej wykonywać) – można stwierdzić, że kolizja nie wystąpiła.

Klasa Gracz

Aby mieć obiekt, którym będzie mógł sterować gracz, należy stworzyć odpowiednią klasę. Napiszemy ją w nowym pliku z klasą **Gracz.cs**.

1 Jak zwykle nasze pisanie rozpoczynamy od stworzenia pól klasy. W tym wypadku potrzebujemy lokalizacji i wymiarów obiektu gracza. Lokalizacja to oczywiście pola **x** i **y**. Wymiary możemy zapisać jako dwa pola **wysokosc** i **szerokosc** lub jako jedno pole przechowujące średnicę (jeśli chcemy, aby obiekt gracza był kulką) albo bok (jeśli chcemy zrobić z niego kwadrat). To zagadnienie można rozwiązać na różne sposoby – w naszym przykładzie zapiszemy dwa wymiary.

2 Konstruktor dla obiektu klasy **Gracz** będzie musiał nadać wartości wszystkim polom klasy. Aby gra dobrze wyglądała, możemy umieścić obiekt gracza w takiej lokalizacji, by jego środek pokrywał się ze środkiem pola gry. Żeby to zrobić, musimy znać wymiary pola gry i wymiary obiektu gracza – w parametrze konstruktora powinny zatem pojawić się takie wartości. Wymiary gracza przypisujemy od razu do pól **szerokosc**

```
public bool czy_kolizja(int x_gracz, int y_gracz, int szer_gracz, int wys_gracz)
{
    if (x+srednica>x_gracz && x < x_gracz + szer_gracz)
    {
        if (y + srednica > y_gracz && y < y_gracz + wys_gracz)
        {
            return true;
        }
    }
}
```

```
public Gracz(int szerokosc_okna, int wysokosc_okna, int wys_gracza, int szer_gracza)
{
    wysokosc = wys_gracza;
    szerokosc = szer_gracza;
    x = (szerokosc_okna - szer_gracza) / 2;
    y = (wysokosc_okna - wys_gracza) / 2;
}
```

i **wysokosc**. Współrzędne **x** i **y** obliczamy natomiast jako różnicę pomiędzy wymiarami okna a wymiarami gracza podzieloną przez dwa. Da nam to takie wartości współrzędnych, że obiekt gracza będzie miał swój środek w miejscu środka pola gry.

3 Budujemy również metodę odpowiedzialną za rysowanie obiektu gracza. Tym razem to kolejna wariacja na temat metody, która pobiera dwa parametry: obiekt graficzny i kolor. Kolor przekazany przez parametr posłuży nam do bezpośredniego utworzenia pędzla przy wywoływaniu metody ***FillElipse*** dla obiektu graficznego. Pozostałe parametry metody ***FillElipse*** to wartości pól klasy – kolejno **x**, **y**, **wysokosc** i **szerokosc**.

```
public void rysuj(Graphics g, Color kol)
{
    g.FillEllipse(new SolidBrush(kol), x, y, szerokosc, wysokosc);
}
```

Gdybyśmy nie chcieli, aby obiekt gracza miał kształt taki jak kulki, możemy zamiast **FillEllipse** użyć funkcji ***FillRectangle***, która narysuje obiekt gracza jako prostokąt.

```
public void rysuj(Graphics g, Color kol)
{
    g.FillRectangle(new SolidBrush(kol), x, y, szerokosc, wysokosc);
}
```

4 Nasza klasa **Gracz** musi mieć też metodę ***przesun***. Metoda ta powinna za parametr przyjąć wartość w typie **string**, która będzie mówiła o tym, w jakim kierunku (w którą stronę) należy się przesunąć. We wnętrzu tej metody należy zbadać wartość parametru i zależnie od niej zmieniać **x** lub **y**, dodając albo odejmując kilka pikseli.

```
public void przesun(string strona)
{
    if (strona == "prawa")
    {
        x = x + 5;
    }
    else if (strona == "lewa")
    {
        x = x - 5;
    }
    else if (strona == "gora")
    {
        y = y - 5;
    }
    else if (strona == "dol")
    {
        y = y + 5;
    }
}
```

Wykorzystanie napisanych klas

Po zdefiniowaniu klas **Kulka** i **Gracz** pora na stworzenie obiektów tych klas i wykorzystanie ich w oknie gry.

1 Pisanie w klasie **Form1** rozpoczynamy od zadeklarowania pola boolowskiego – **czy_gra_aktywna**. W tym projekcie, jak w poprzednich, będziemy wyświetlać poprzez instrukcje w zdarzeniu **Timera** różne treści zależnie od tego, czy gra została włączona, czy też jeszcze nie.

```
public partial class Form1 : Form
{
    bool czy_gra_aktywna = false;

    public Form1()
    {
        InitializeComponent();
    }
```

2 Drugi krok to powrót do okna modelowania i – poprzez dwukrotne kliknięcie na dodany do projektu **Timer** – wygenerowanie metody odpowiedzialnej za zdarzenie **Timera**. W jej treści – podobnie jak w poprzednich projektach, powinna znaleźć się instrukcja warunkowa **if**, która będzia sprawdzać, czy gra jest aktywna. Jeśli gra nie jest aktywna, powinien pojawić się na panelu napis, mówiący o tym, jak włączyć grę. Tym razem pojawi się też informacja o tym, żeby gracz wpisał w polu tekstowym liczbę kulek, jakie chce mieć w grze. Warto pamiętać o dobraniu odpowiednich współrzędnych, w których umieszczamy napis – powinny pasować do wymiarów Panelu.

```
private void timer1_Tick(object sender, EventArgs e)
{
    if (czy_gra_aktywna)
    {

    }
    else
    {
        FontFamily fFamily = new FontFamily("Arial");
        Font f = new Font(fFamily, 15);
        Brush b = new SolidBrush(Color.Black);
        panel1.CreateGraphics().DrawString("Podaj liczbę kulek,", f, b, 120, 135);
        panel1.CreateGraphics().DrawString("następnie naciśnij Start", f, b, 100, 165);
    }
}
```

3 Zanim przejdziemy do zapisania tego, co ma się wyświetlać, kiedy gra jest aktywna, musimy zadeklarować odpowiednie pola dla klasy **Form1**. Jednym z nich będzie obiekt typu **Random**, który wykorzystamy

```
public partial class Form1 : Form
{
    bool czy_gra_aktywna = false;
    Random r = new Random();
    Kulka[] k = new Kulka[100];
    Gracz g;
    int ile_kulek;
```

w konstruktorze kulek. Kulek będzie dużo – ile? Teraz tego nie wiemy. Zakładamy, że maksymalnie może być ich **100**. Deklarujemy więc tablicę składającą się ze stu obiektów klasy **Kulka**. Deklarujemy też obiekt gracza. Przyda nam się również pole mówiące o tym, ile kulek powinno być w grze. Pole to swoją wartość dostanie po wpisaniu przez gracza liczby w pole tekstowe i naciśnięciu przez niego przycisku **Start**.

4 Aby przejść dalej, będziemy musieli znać liczbę kulek. Pole **ile_kulek** swoją wartość otrzyma po kliknięciu na przycisk **Start**. Przenosimy się do okna modelowania i dwukrotnym kliknięciem na przycisk na modelu okna generujemy odpowiednią metodę. W jej treści przypisujemy do **ile_kulek** zrzutowaną na typ **Integer** wartość właściwości **Text** z dodanego do okna gry Texboxa.

```
private void button1_Click(object sender, EventArgs e)
{
    ile_kulek = int.Parse(textBox1.Text);
    if (ile_kulek>100) { ile_kulek = 100; }
}
```

5 W kolejnej linii możemy sprawdzić za pomocą instrukcji warunkowej **if**, czy wartość, jaką tam umieściliśmy, nie jest większa niż **100**. Jeśli jest – zmieniamy wartość **ile_kulek** na **100** (dlatego że taki jest wymiar tablicy obiektów typu **Kulka**; nie możemy umieścić w tej tablicy więcej obiektów).

```
private void button1_Click(object sender, EventArgs e)
{
    ile_kulek = int.Parse(textBox1.Text);
    if (ile_kulek>100) { ile_kulek = 100; }
    czy_gra_aktywna = true;
```

6 Oprócz tego do pola **czy_gra_aktywna** wpisujemy wartość **true** – aby wystartować grę.

7 Zanim jednak gra faktycznie się rozpocznie, musimy utworzyć obiekty: gracza oraz kulki. Zaczynamy od wywołania konstruktora dla obiektu gracza. W jego parametrach podajemy wymiary pola gry i wymiary samego obiektu gracza.

```
private void button1_Click(object sender, EventArgs e)
{
    ile_kulek = int.Parse(textBox1.Text);
    if (ile_kulek>100) { ile_kulek = 100; }
    czy_gra_aktywna = true;
    g = new Gracz(panel1.Width, panel1.Height, 20, 20);
}
```

8 Dalej należy wywołać konstruktor dla każdego obiektu **Kulka** z tablicy kulek. Zrobimy to poprzez pętlę **for** **A**, która wykona się tyle razy, ile wynosi wartość pola **ile_kulek**. Wewnątrz pętli dla i-tego elementu tablicy wywołujemy konstruktor **B** obiektu klasy **Kulka**.

```
private void button1_Click(object sender, EventArgs e)
{
    ile_kulek = int.Parse(textBox1.Text);
    if (ile_kulek>100) { ile_kulek = 100; }
    czy_gra_aktywna = true;
    g = new Gracz(panel1.Width, panel1.Height, 20, 20);
  A for (int i = 0; i < ile_kulek; i++)
    {
      B k[i] = new Kulka(panel1.Width, panel1.Height, r);
    }
}
```

9 Możemy teraz powrócić do metody przypisanej do zdarzenia **Timera**. Zajmiemy się w niej sytuacją, w której **czy_gra_aktywna** ma wartość **true**. Po pierwsze, zamazujemy poprzez funkcję **Clear** obszar pola gry. Po drugie, rysujemy na polu gry obiekt gracza poprzez metodę ***rysuj***.

```
if (czy_gra_aktywna)
{
    panel1.CreateGraphics().Clear(Color.LightBlue);
    g.rysuj(panel1.CreateGraphics(), Color.Red);
}
```

```
if (czy_gra_aktywna)
{
    panel1.CreateGraphics().Clear(Color.LightBlue);
    g.rysuj(panel1.CreateGraphics(), Color.Red);
    for (int i = 0; i < ile_kulek; i++)
    {
        k[i].rysuj(panel1.CreateGraphics()); C
        k[i].przesun(panel1.Width, panel1.Height); D
    }
}
else
```

10 Następnie należy wyrysować wszystkie kulki. Aby to zrobić, musimy stworzyć pętlę **for**, która pozwoli na odnoszenie się do kolejnych elementów tablicy kulek. Wewnątrz tej pętli zarówno rysujemy **C**, jak i od razu przesuwamy **D** kulki w nowe miejsce.

11 Możemy jeszcze przejść do metody odpowiedzialnej za kliknięcie na przycisk startujący. W nim możemy wyłączyć **Textbox** i **Button** na czas, gdy gra jest aktywna, przypisując do ich właściwości **Enabled** wartość **false**.

12 Ponownie włączymy możliwość wpisania liczby kulek i wystartowania gry, gdy zajdzie kolizja pomiędzy graczem a którąś z kulek. Sprawdzanie kolizji zapisujemy poprzez wywołanie metody ***kolizja*** dla kulki. Musimy ją jednak wywołać dla każdej kulki z tablicy. Robimy to poprzez dodanie instrukcji warunkowej **if** w pętli **for** znajdującej się w metodzie odpowiedzialnej za zdarzenie **Timera**.

13 Jeśli kolizja występuje, zmieniamy wartość pola **czy_gra_aktywna** na **false**, a także zmieniamy właściwość **Enabled** w **Buttonie** i **Textboxie** na **true** – aby możliwe było ponowne uruchomienie gry.

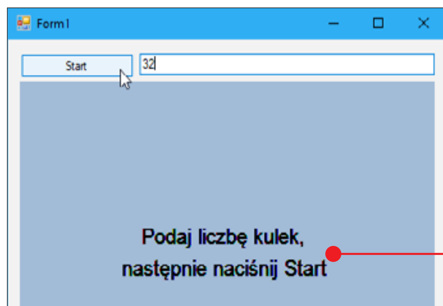
```
if (czy_gra_aktywna)
{
    panel1.CreateGraphics().Clear(Color.LightBlue);
    g.rysuj(panel1.CreateGraphics(), Color.Red);
    for (int i = 0; i < ile_kulek; i++)
    {
        k[i].rysuj(panel1.CreateGraphics());
        k[i].przesun(panel1.Width, panel1.Height);
        if (k[i].czy_kolizja(g.x, g.y, g.szerokosc, g.wysokosc))
        {
            czy_gra_aktywna = false;
            textBox1.Enabled = true;
            button1.Enabled = true;
        }
    }
}
```

14 Aby nasza gra działała, brakuje już tylko możliwości sterowania obiektem gracza. W tym celu przenosimy się do okna modelowania gry. Tam w sekcji **Właściwości** dla obiektu **Form1** przechodzimy do zdarzeń i spośród nich wybieramy **KeyDown** – czyli wciśnięcie klawisza na klawiaturze. Dwukrotne kliknięcie na tę opcję spowoduje wygenerowanie metody uruchamianej wraz z naciśnięciem dowolnego klawisza z klawiatury. My w treści tej metody – podobnie jak w poprzednich projektach – sprawdzamy, który z klawiszy został naciśnięty. Jeśli były to strzałki, to zależnie od tego, którą strzałkę naciśnięto, wywołujemy metodę **przesun** dla obiektu gracza z odpowiednim parametrem tekstowym mówiącym o stronie, w którą chcemy przesunąć obiekt.

Testowanie

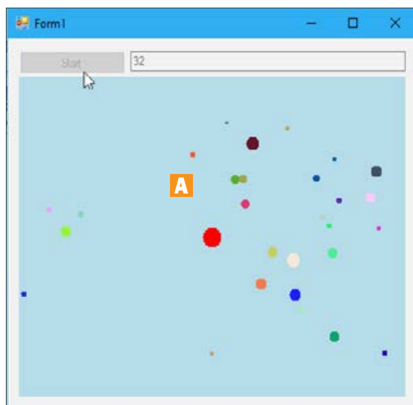
Wygląda na to, że nasza gra jest gotowa. Możemy ją teraz przetestować.

1 Po uruchomieniu gry widzimy napisy z informacją, jak rozpocząć rozgrywkę.

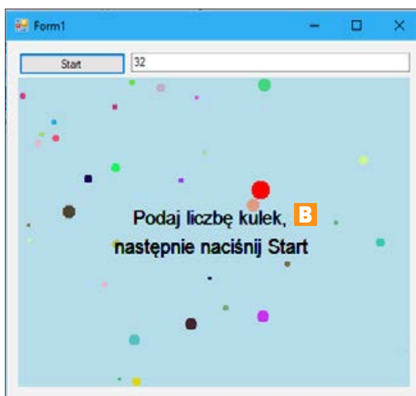


2 Po wpisaniu liczby kulek w pole tekstowe należy nacisnąć przycisk **Start**. Na środku powinna pokazać się duża czerwona kulka **A**, będąca obiektem gracza, i mniejsze kulki, przed którymi gracz musi uciekać. Widzimy również, że pole tekstowe i przycisk **Start** nie są aktywne, gdy trwa rozgrywka.

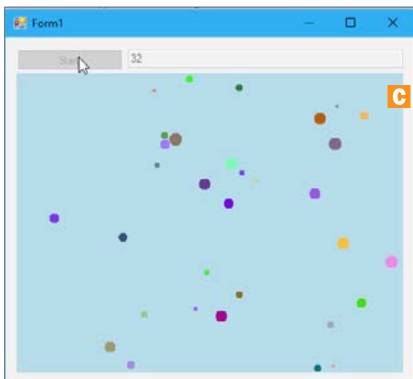
3 Kiedy obiekt gracza zderzy się z którąś z kulek, pojawia się ponownie napis **B**, który był widoczny na początku, po uruchomieniu gry. Kulki przestają się poruszać, gra



zatrzymuje się. Ponownie aktywny staje się **Textbox** i przycisk **Start**.



4 Prowadząc rozgrywkę, szybko możemy zauważyć, że naszą grę łatwo „obejść". Można wyjść poza okno i unikać w ten sposób zderzenia z kulkami **C**. Aby zlikwidować ten problem, musimy zmodyfikować metodę ***przesun***. Zlikwidowanie możliwości wyjścia poza lewą i górną krawędź nie



```
else if (strona == "lewa" && x > 0)
{
    x = x - 5;
}
```

jest trudne W metodzie ***przesun*** klasy **Gracz** trzeba dodać kolejne warunki do instrukcji warunkowej. Gdy sprawdzamy, czy strona, w którą chcemy przesunąć obiekt gracza, to lewa, dodajemy drugi warunek – sprawdzający, czy współrzędna **x** gracza jest większa niż **0** – tylko wtedy możemy zmniejszyć **x**.

5 Podobnie działamy, gdy strona, w którą chcemy przesunąć obiekt gracza, to góra – dodajemy drugi warunek, który sprawdza, czy współrzędna **y** obiektu gracza jest większa niż **0**. Tylko wtedy możemy zmniejszyć współrzędną **y**.

```
else if (strona == "gora" && y > 0)
{
    y = y - 5;
}
```

6 Inaczej sprawa ma się z pozostałymi dwoma krawędziami pola gry. Musimy znać wymiary pola gry. Aby zapisać je w naszej klasie, musimy dodać do niej dwa pola – odpowiadające właśnie wymiarom okna gry.

```
class Gracz
{
        public int y;
        public int x;
        public int szerokosc;
        public int wysokosc;
        public int szer_okna;
        public int wys_okna;
```

7 Musimy zmodyfikować też konstruktor obiektu tej klasy. Przyjmował on już co prawda wymiary pola gry w parametrach – jednak wykorzystywał je w innym celu. Teraz dopiszemy do konstruktora zapisanie dwóch wartości z parametrów do pól.

```
public Gracz(int szerokosc_okna, int wysokosc_okna, int wys_gracza, int szer_gracza)
{
    wysokosc = wys_gracza;
    szerokosc = szer_gracza;
    x = (szerokosc_okna - szer_gracza) / 2;
    y = (wysokosc_okna - wys_gracza) / 2;
    szer_okna = szerokosc_okna;
    wys_okna = wysokosc_okna;
}
```

8 Mając już wartości w dwóch nowych polach naszej klasy **Gracz**, możemy wykorzystać je w metodzie ***przesun***. Musimy dodać warunek do instrukcji **if**, kiedy chcemy przesuwać obiekt gracza w prawą stronę. Sprawdzamy w nim, czy współrzędna **x** ma taką wartość, aby obiekt gracza mieścił się w szerokości pola gry.

```
public void przesun(string strona)
{
    if (strona == "prawa" && x+szerokosc <szer_okna)
    {
        x = x + 5;
    }
```

9 Analogicznie do przesuwania się w prawą stronę dodajemy warunek do instrukcji mówiącej o przesuwaniu się w dół.

```
else if (strona == "dol" && y + wysokosc < wys_okna)
{
    y = y + 5;
}
```

System liczenia punktów

Nasza gra działa, jednak trudno w niej porównywać osiągnięcia graczy. W naszej grze brakuje systemu liczenia punktów.

1 Dodajemy pole **punkty** w typie **Integer**, dla klasy **Form1**.

```
public partial class Form1 : Form
{
    bool czy_gra_aktywna = false;
    Random r = new Random();
    Kulka[] k = new Kulka[100];
    Gracz g;
    int ile_kulek;
    int punkty;
    public Form1()
    {
```

2 W metodzie odpowiadającej za kliknięcie na przycisk startujący grę ustawiamy wartość pola **punkty** **A** na **0**.

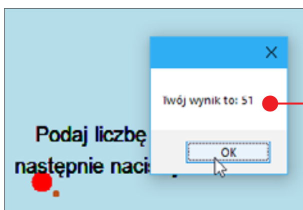
3 W metodzie wywoływanej przez **Timer**, kiedy gra jest aktywna – zwiększamy wartość zmiennej **punkty** **B**.

```
private void button1_Click(object sender, EventArgs e)
{
    ile_kulek = int.Parse(textBox1.Text);
    if (ile_kulek>100) { ile_kulek = 100; }
    czy_gra_aktywna = true;
    g = new Gracz(panel1.Width, panel1.Height, 20, 20);
    for (int i = 0; i < ile_kulek; i++)
    {
        k[i] = new Kulka(panel1.Width, panel1.Height, r);
    }
    textBox1.Enabled = false;
    button1.Enabled = false;
    punkty = 0; A
}
```

4 Aby przekazać graczowi liczbę uzyskanych przez niego punktów, przechodzimy do miejsca w kodzie, w którym wykrywaliśmy kolizję kulki z obiektem gracza. Dodajemy tam polecenie ***MessageBox.Show()***, którego wywołanie skutkuje wyświetleniem okna z informacją. Informację wpisujemy w parametrze polecenia. Informacja to ciąg znaków, który może składać się z tekstu i wartości pola punkty, którą trzeba jednak przekonwertować (**Convert.ToString**) na typ **string**.

```
if (k[i].czy_kolizja(g.x, g.y, g.szerokosc, g.wysokosc))
{
    czy_gra_aktywna = false;
    textBox1.Enabled = true;
    button1.Enabled = true;
    MessageBox.Show("Twój wynik to: " + Convert.ToString(punkty));
}
```

5 Po tych zmianach, gdy gracz przegra, pojawi się okno z informacją o zdobytej liczbie punktów.




```
if (czy_gra_aktywna)
{
    panel1.CreateGraphics().Clear(Color.LightBlue);
    g.rysuj(panel1.CreateGraphics(), Color.Red);
    for (int i = 0; i < ile_kulek; i++)
    {
        k[i].rysuj(panel1.CreateGraphics());
        k[i].przesun(panel1.Width, panel1.Height);
        if (k[i].czy_kolizja(g.x, g.y, g.szerokosc, g.wysokosc))
        {
            czy_gra_aktywna = false;
            textBox1.Enabled = true;
            button1.Enabled = true;
        }
    }
    punkty = punkty + 1; B
}
```

DOKOŃCZ SAMODZIELNIE*

Nasza gra działa już całkiem sprawnie – nie oznacza to jednak, że nie może działać jeszcze lepiej. Warto jeszcze poeksperymentować, na przykład można:

1. Sprawić, aby liczba dodawanych punktów była różna – zależnie od poziomu trudności gry (o poziomie trudności mówi liczba kulek)
2. Sprawić, aby wraz z czasem trwania gry rosła liczba kulek, których unikać musi gracz.

*__Podpowiedź:__ Zadania te są bardziej logiczne niż programistyczne: **1.** Zmienna **punkty** może zwiększać swoją wartość nie o 1, ale o wartość zmiennej **ile_kulek** – i już mamy punktowanie uzależnione od poziomu trudności. **2.** To zadanie też można rozwiązać, manipulując jedynie liczbą kulek.

DODATKOWE NARZĘDZIA DO TWORZENIA APLIKACJI NA ANDROIDA ZNAJDZIESZ NA DVD I W KŚ+

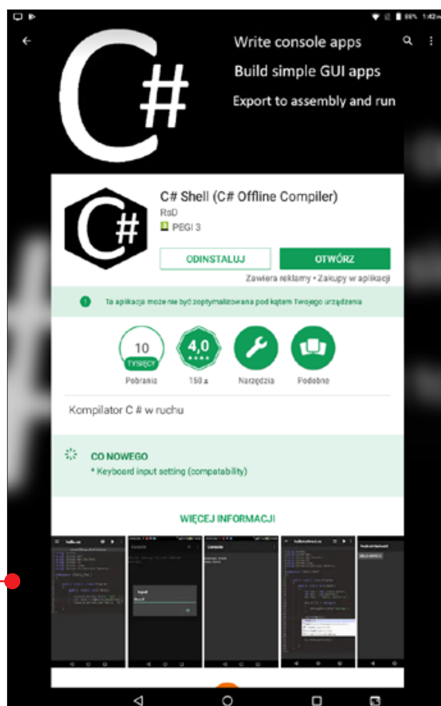
5 Programowanie mobilne w C#

Wiele osób, decydując się na rozpoczęcie przygody z programowaniem, myśli o tym, aby tworzyć gry i aplikacje na urządzenia mobilne. Tworzenie aplikacji mobilnych może być bardzo interesujące. Równie interesujące jest jednak samo programowanie bezpośrednio na urządzeniu mobilnym – to też jest możliwe!



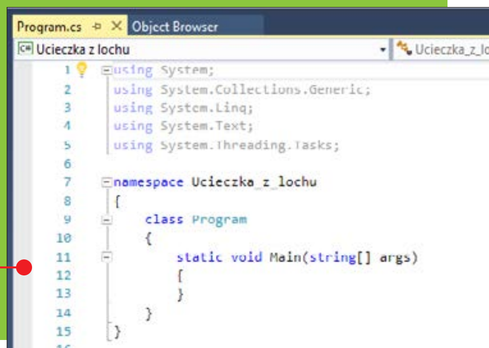
Wybór narzędzia

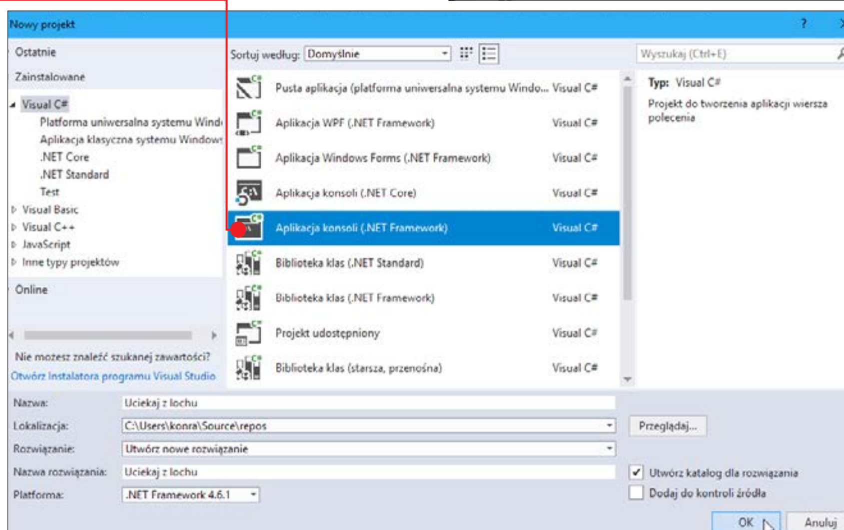
Programując, nie warto ograniczać się do jednego języka, narzędzia – a nawet jednego rodzaju urządzeń. Microsoft Visual Studio – to IDE przeznaczone dla komputerów PC z systemem Windows. Programować można jednak także na smartfonach i tabletach. W sklepie z aplikacjami na urządzeniu mobilnym wystarczy wyszukać kompilator języka C# (C# Compiler) – jak choćby **C# Shell** w Sklepie Play. Na kolejnych stronach przeczytamy, jak za pomocą tej aplikacji zrobić prostą grę tekstową.



PROGRAMOWANIE MOBILNE TAKŻE NA KOMPUTERZE

Przedstawione w tym rozdziale zadanie z powodzeniem można zrealizować także bez urządzenia mobilnego – na komputerze, wykorzystując, jak w poprzednich rozdziałach, **MS Visual Studio**. Ponieważ opisywana gra jest grą tekstową, tworząc nowy projekt, nie będziemy korzystać ze schematu Windows Forms Application, ale **Aplikacja konsoli (.NET Framework)**. Po utworzeniu takiego projektu jesteśmy od razu przenoszeni do kodu – bez fazy projektowania okna (które w tego typu aplikacjach zastąpione jest prostą konsolą, w której wypisujemy i wczytujemy dane). Kod, jaki dostajemy na start, jest prawie identyczny z kodem z aplikacji mobilnej **C# Shell**, dlatego kroki opisane na kolejnych stronach można wykonywać także na komputerze.



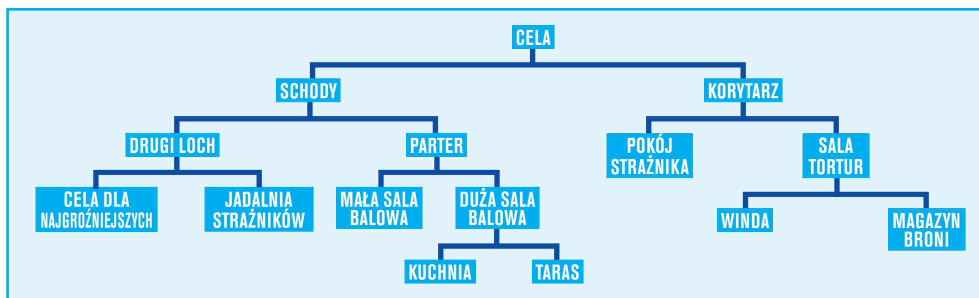


Projektowanie świata gry

Przedstawiona aplikacja jest przygodową grą tekstową. Każdy Czytelnik może stworzyć ją według własnego pomysłu. Pomysł przedstawiony w tym rozdziale jest następujący: gracz trafia do lochu w zamku, z którego musi uciec, przemieszczając kolejne pomieszczenia. Będąc w danej komnacie, może przejść do jednej z dwóch kolejnych lub

zawrócić. Zadaniem gracza jest takie wybieranie pomieszczeń, aby dojść do określonego wcześniej celu. Z części sal nie ma wyjścia – wejście do nich oznacza przegraną.
W uproszczeniu nasza gra składa się ze zbioru pomieszczeń połączonych ze sobą przejściami. W tego typu projektach najważniejsze jest wymyślenie drogi, po której będzie poruszał się gracz, a także opisanie wszystkich możliwych lokalizacji. Gracz, przechodząc przez kolejne komnaty i lochy, stworzy prawdziwą opowieść. Projektując

POMIESZCZENIE	OPIS
Cela	Jakiś czas temu pomylono cię z groźnym przestępcą i wtrącono do celi w lochu. Dziś nadarzyła się okazja do ucieczki. Strażnik nie domknął kłódki. Wychodzisz z celi.
Schody	Wejście na schody to chyba dobry pomysł. Tylko gdzie szukać ucieczki? Być może na górze – może uda się wyjść głównymi wrotami do zamku? A może iść w dół i szukać sekretnego przejścia podziemnego?
Korytarz	Idziesz w głąb lochu. Jest ciemno i zimno. Może to chłód z tajemnego przejścia? Na razie jednak nic takiego nie widać. Są za to kolejne pomieszczenia. Może w nich znajdziesz drogę ucieczki?
Drugi loch	Ten loch jest jeszcze głębiej pod ziemią, niż cię więziono. Jest tu jeszcze zimniej. Nic nie wskazuje na to, żeby gdzieś tu znajdowało się sekretne przejście, którym możesz uciec. Są tu za to kolejne pomieszczenia.
Cela dla najgroźniejszych przestępców	O nie! Trafiasz do celi dla najgroźniejszych przestępców, akurat gdy przebywa tam Lortus Hurtus! To prawdziwy krwiopijca, przed którym nie ustrzeże się nikt. Nawet ty. Wygląda na to, że będziesz jego kolejną ofiarą...
Jadalnia strażników	Wejście do jadalni strażników, akurat gdy jedzą obiad, to nie jest dobry pomysł. Nici z ucieczki. Strażnicy, mimo że skupieni na jedzeniu, od razu cię zauważają.
Mała sala balowa	W zamku trwa bal, jednak nie w tej sali. Tutaj jest za to osobista służba króla. Jego najwierniejsi podwładni. Oni, jak nikt inny, dbają o jego bezpieczeństwo i nie pozwolą na to, by cokolwiek mu zagroziło. Dla nich nie jest ważne, czy uciekasz z lochu – i tak jesteś kimś obcym, jesteś wrogiem i trzeba z tobą skończyć...
Duża sala balowa	Trwa bal! Jest tu wiele osób, a wszyscy w pięknych balowych strojach. Od razu widać, że uciekasz z lochu, ponieważ masz na sobie stare, podarte i śmierdzące ubrania. Szybko – biegnij, nim ktokolwiek cię zobaczy.
Parter	W końcu nie jesteś pod ziemią. Widzisz wejście, jednak są przy nim strażnicy. Na szczęście stoją tyłem do ciebie. Schowaj się w jakimś pomieszczeniu, zanim się obrócą!
Kuchnia	Przy samym wejściu do kuchni znajdujesz wór, w którym kucharz trzymał swoje ubrania. Wykorzystujesz zamieszanie, jakie panuje w królewskiej kuchni, i zakładasz czyste ubrania. W nich możesz już bez trudu wyjść z zamku. Wyglądasz teraz jak wolny człowiek i stajesz się nim naprawdę, wychodząc tylnym wejściem, którym wnoszona jest żywność!
Taras	Widzisz ogród! Jesteś na świeżym powietrzu! Niestety, twój plan ucieczki przez ogród nie może się powieść, ponieważ taras jest doskonale strzeżony przez królewskich strażników. Tędy nie uciekniesz. A oni już cię widzą... To nie był dobry pomysł, aby tu wchodzić.
Pokój strażnika	Uciekać z lochu i wejść do pokoju strażnika? To brzmi absurdalnie i jest absurdalne. W ten sposób nigdy nie uciekniesz. Strażnik już cię schwytał.
Sala tortur	Ciekawe miejsce! Pod warunkiem że nie trafiasz tam razem z królewskim katem, który może położyć cię na łożu do rozciągania. Na szczęście teraz w sali nikogo nie ma, a ty dostrzegasz dla siebie szansę, aby iść dalej.
Winda	Winda, a raczej dźwig, który służy do wprowadzania do lochu nadwornego lwa. Pech chciał, że zwierzę jest w środku, kiedy tam wchodzisz. To nie może być dla ciebie miłe spotkanie.
Magazyn broni	Tu strażnicy trzymają swoją broń, ale też narzędzia tortur. Pech chciał, że właśnie zakupiono nowe szpikulce. Stajesz się idealnym kandydatem, aby je na tobie przetestować.



taką grę, najlepiej jest przedstawić pomieszczenia w formie drzewa ●. W zaprezentowanym skrypcie zostały opisane pomieszczenia z przedstawionego powyżej drzewa.
Gra ta ma jeden ogromny atut – każdy, pisząc jej własną wersję, może wykorzystać swoje pomysły i dodać znacznie więcej pomieszczeń. Rozmiar świata gry zależy tylko od inwencji twórczej osoby piszącej – w grze może być kilkadziesiąt, kilkaset, a nawet kilka tysięcy pomieszczeń! Ogranicza nas tylko nasza pomysłowość.
W opisanym przykładzie gracz rozpoczyna przygodę w celi, z której może iść dalej na schody lub przemierzać korytarz. Jeśli wybierze schody, może iść w górę lub w dół, a jeśli wybierze wędrówkę korytarzem, może przemierzać kolejne pomieszczenia.
Projektując własne drzewo pomieszczeń, trzeba zastanowić się nad opisami, które będą wyświetlane dla gracza. To one są głównym elementem gry i to z nich gracz będzie dowiadywał się, co się wokół niego dzieje, ponieważ gra pozbawiona jest interfejsu graficznego.
Tworząc opisy, dobrze jest zaznaczyć na drzewie pomieszczeń, skąd nie ma wyjścia i gdzie gra kończy się przegraną, a gdzie zwycięstwem gracza. W przykładowej grze zwycięstwem będzie wejście do kuchni.

Zaczynamy pisanie – klasa Pomieszczenie

Kiedy przygotowaliśmy już scenariusz gry i opisy kolejnych pomieszczeń, możemy przejść do programowania.

1 Po uruchomieniu mobilnego programu C# Shell mamy niewielki skrypt zawierający klasę **Program** i jej funkcję główną **Main** ●, która wykona się przy uruchomieniu gry.

2 Co jednak należy wpisać w funkcji **Main**? Nie możemy zapomnieć o tym, jak duży potencjał daje nam poznany niedawno paradygmat programowania obiektowego. Niezależnie od tego, czy tworzymy grę na komputerze PC, czy na urządzeniu mobilnym,

ucieczka z lochu.cs

← → ↑ ↓ Copy selection Paste at cursor

```
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Linq;
using System.Collections.Generic;

namespace CSharp_Shell
{
    public static class Program
    {
        public static void Main()
        {

        }
    }
}
```

możemy tworzyć ją w zgodzie z poznanym paradygmatem. Zgodnie z nim należałoby stworzyć nową klasę, a w funkcji **Main** utworzyć jej obiekty i nimi posługiwać się w grze. Tak właśnie zrobimy. Stworzymy klasę **Pomieszczenie**. Jej definicję umieścimy w tym samym pliku, który mamy aktualnie otwarty. Pisanie rozpoczynamy po zamknięciu nawiasu klamrowego klasy **Program**.

3 Co łączy wszystkie pomieszczenia? To, że mają one nazwę i opis. Część pomieszczeń ma także wyjścia prowadzące do innych pomieszczeń. To wszystko należy zatem opisać jako pola klasy. Pola **nazwa** i **opis** – to teksty, czyli ciągi znaków. Zadeklarujemy je zatem w typie **string**.

ucieczka z lochu.cs

← → ↑ ↓ Copy selection Paste at cursor

```
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Linq;
using System.Collections.Generic;

namespace CSharp_Shell
{

    public static class Program
    {
        public static void Main()
        {

        }

    }

    public class Pomieszczenie
    {
        public string nazwa;
        public string opis;
        public Pomieszczenie lewo;
        public Pomieszczenie prawo;
    }
}
```

4 Zadeklarujemy również dwa pola – **lewo** i **prawo**, które będą odnosiły się do innych obiektów klasy **Pomieszczenie** i będą wskazywać na sale, do których można przejść, będąc w opisywanym pomieszczeniu.

5 Konstruktor obiektu klasy **Pomieszczenie** będzie przyjmował tylko dwa

```
public class Pomieszczenie
{
    public string nazwa;
    public string opis;
    public Pomieszczenie lewo;
    public Pomieszczenie prawo;
    public Pomieszczenie(string n, string o)
    {
        this.nazwa = n;
        this.opis = o;
    }
}
```

parametry – obydwa w typie **string**. Będą one przypisywane do nazwy i do opisu.

6 Co jednak z pozostałymi dwoma polami klasy? One swoich wartości nie dostaną poprzez konstruktor, ale poprzez specjalną metodę **ustaw**. Ona za parametr przyjmie dwa obiekty klasy **Pomieszczenie** i przypisze je do pól. Metoda ta będzie wywoływana tylko dla niektórych pomieszczeń – tylko dla tych, przez które można przejść.

ucieczka z lochu.cs

← → ↑ ↓ Copy selection Paste at cursor

```
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Linq;
using System.Collections.Generic;

namespace CSharp_Shell
{

    public static class Program
    {
        public static void Main()
        {

        }
    }

    public class Pomieszczenie
    {
        public string nazwa;
        public string opis;
        public Pomieszczenie lewo;
        public Pomieszczenie prawo;
        public Pomieszczenie(string n, string o)
        {
            this.nazwa = n;
            this.opis = o;
        }
        public void ustaw(Pomieszczenie p, Pomieszczenie l)
        {
            this.lewo = l;
            this.prawo = p;
        }
    }
}
```

Funkcja główna Main

Kiedy napisaliśmy już klasę **Pomieszczenie**, przyszedł czas na utworzenie jej obiektów.

1 Przenosimy się do funkcji głównej programu – czyli **Main**. To właśnie tam zadeklarujemy obiekty klasy **Pomieszczenie**. Teraz przyda nam się tabela, w której znajdują się opisy pomieszczeń. Każde pomieszczenie z tabeli będzie teraz zadeklarowane jako obiekt klasy **Pomieszczenie**. Dla każdego obiektu wywołany powinien być konstruktor, w którego parametrach wpisujemy kolejno nazwę pomieszczenia i jego opis (z tabeli). Mając 15 pomieszczeń, deklarujemy 15 obiektów.

2 Następnie dla pomieszczeń, z których przewidziano przejścia, wywołujemy metodę **ustaw**. W jej parametrach podajemy nazwy obiektów odpowiadających

```
Pomieszczenie pokoj_straznika = new Pomieszczenie("Pokój strażnika
Pomieszczenie sala_tortur = new Pomieszczenie("Sala tortur", "Piękı
Pomieszczenie winda = new Pomieszczenie("Winda", "Winda, a raczej 
Pomieszczenie magazyn_broni = new Pomieszczenie("Magazyn broni", "T

cela.ustaw(schody, korytarz);
korytarz.ustaw(pokoj_straznika, sala_tortur);
sala_tortur.ustaw(winda, magazyn_broni);
schody.ustaw(drugi_loch, parter);
drugi_loch.ustaw(cela_dla_najgrozniejszych, jadalnia_straznikow);
parter.ustaw(mala_balowa, duza_balowa);
duza_balowa.ustaw(kuchnia, taras);
```

```
← → ↑ ↓ Copy selection Paste at cursor

    public static class Program
    {
        public static void Main()
        {
            Pomieszczenie cela = new Pomieszczenie("cela", "Jakiś czas temu pomylono Cię z groźny
             Pomieszczenie schody = new Pomieszczenie("Schody", "Wejście na schody to chyba dobry
             Pomieszczenie korytarz = new Pomieszczenie("Korytarz", "Idziesz w głąb lochu. Jest c
             Pomieszczenie drugi_loch = new Pomieszczenie("Drugi loch", "Ten loch jest jeszcze gł
             Pomieszczenie cela_dla_najgrozniejszych = new Pomieszczenie("Cela dla najgroźniejszy
             Pomieszczenie jadalnia_straznikow = new Pomieszczenie("Jadalnia strażników", "Wejści
             Pomieszczenie mala_balowa = new Pomieszczenie("Mała sala balowa", "W zamku trwa bal,
             Pomieszczenie duza_balowa = new Pomieszczenie("Duża sala balowa", "Trwa bal!Jest tu 
             Pomieszczenie parter = new Pomieszczenie("Parter", "W końcu nie jesteś pod ziemią. W
             Pomieszczenie kuchnia = new Pomieszczenie("Kuchnia", "Przy samym wejściu do kuchni z
             Pomieszczenie taras = new Pomieszczenie("Taras", "Widzisz ogród! Jesteś na świeżym p
             Pomieszczenie pokoj_straznika = new Pomieszczenie("Pokój strażnika", "Uciekać z loch
             Pomieszczenie sala_tortur = new Pomieszczenie("Sala tortur", "Piękne miejsce! Dopóki
             Pomieszczenie winda = new Pomieszczenie("Winda", "Winda, a raczej dźwig, który służy
             Pomieszczenie magazyn_broni = new Pomieszczenie("Magazyn broni", "Tu strażnicy trzym


        }

    }

    public class Pomieszczenie
    {
        public string nazwa;
        public string opis;
        public Pomieszczenie lewo;
        public Pomieszczenie prawo;
        public Pomieszczenie(string n, string o)
        {
            this.nazwa = n;
            this.opis = o;
        }
        public void ustaw(Pomieszczenie p, Pomieszczenie l)
        {
            this.lewo = l;
            this.prawo = p;
        }
    }
}
```

pomieszczeniom, do których można przejść z pomieszczenia, dla którego metoda jest wywoływana.

3 Dalej deklarujemy kolejne dwa pomieszczenia – **gdzie** i **poprzednie**. Dla nich nie wywołujemy jednak konstruktora, bo nie

```
parter.ustaw(mala_balowa, duza_balowa
duza_balowa.ustaw(kuchnia, taras);

Pomieszczenie gdzie = cela;
Pomieszczenie poprzednie = cela;
```

tworzą one nowych obiektów. Będą one nam służyły do przechowania już istniejących pomieszczeń. Pierwsze będzie przechowywać pomieszczenie, w którym aktualnie znajduje się gracz. Drugie będzie przechowywać pomieszczenie, w którym gracz był poprzednio – aby móc do niego wrócić.

4 Dalej budujemy pętlę **while** – o ile pętla **for** wykonuje się określoną liczbę razy, to pętla **while** pozwala nam na wykonywanie zestawu instrukcji, aż osiągnięty zostanie pewien cel. Pętla ta w nawiasie przyjmuje warunek niczym **if**. Dopóki warunek ten jest prawdziwy, pętla będzie się wykonywać. Kiedy warunek przestanie być prawdziwy, pętla skończy się powtarzać. Naszym warunkiem na wykonywanie pętli będzie to, że nie doszliśmy jeszcze do pomieszczenia docelowego – zgodnie ze schematem docelowym pomieszczeniem jest kuchnia. Zatem jeśli w zmiennej przechowującej pomieszczenie, w którym jest gracz, nie będzie aktualnie obiektu kuchnia, pętla będzie się wykonywać, a kiedy gracz wejdzie do kuchni, zakończy pętlę.

5 Wewnątrz pętli będziemy pisać uniwersalny skrypt, który będzie odnosił się do pomieszczenia, w którym jest gracz, wyświetli jego opis i pozwoli na przejście dalej. Zaczynamy zatem od wypisania poleceniem **Console.WriteLine** nazwy i opisu pomieszczenia. Dodatkowo możemy łączyć ze sobą ciągi znaków – poprzez plusy pomiędzy nimi.

```
Copy selection   Paste at cursor

            drugi_loch.ustaw(cela_dla_najgrozniejszych, jadalnia_straznikow);
            parter.ustaw(mala_balowa, duza_balowa);
            duza_balowa.ustaw(kuchnia, taras);

            Pomieszczenie gdzie = cela;
            Pomieszczenie poprzednie = cela;

            while (gdzie != kuchnia)
            {
                Console.WriteLine("Jestes w: " + gdzie.nazwa + ". " + gdzie.opis);
                if (gdzie.lewo != null && gdzie.prawo != null) {
                    Console.WriteLine("Możesz iść do: 1." + gdzie.lewo.nazwa + " 2." + gdzie.prawo.nazwa + " 3. Zawróć");   A
                    int wybor = int.Parse(Console.ReadLine());   B
                    if (wybor == 1)   C
                    {
                        poprzednie = gdzie;
                        gdzie = gdzie.lewo;
                    }
                    else if (wybor == 2)
                    {
                        poprzednie = gdzie;
                        gdzie = gdzie.prawo;
                    }
                    else
                    {
                        gdzie = poprzednie;
                    }
                }
                else
                {
                    Console.WriteLine("Twoja przygoda kończy się właśnie tutaj.");
                    Console.WriteLine("Spróbuj jeszcze raz, zaczynając od początku.");
                    gdzie = cela;
                }
            }
            Console.WriteLine("Gratulacje! Udało CI się uciec!");   D
        }

    }
```

6 Kolejna część to instrukcja **if**, która sprawdza, czy dla pomieszczenia, w którym jest gracz, można będzie wypisać pomieszczenia, do których można przejść. Jest to możliwe, tylko gdy pomieszczenie, w którym gracz się znajduje, ma przypisane obiekty do pól **lewo** i **prawo**. Jeśli nie ma, znajduje się tam wartość **null** (czyli puste).

7 Zajmijmy się najpierw sekcją **else** wspomnianej instrukcji. Kiedy nie ma obiektów pod **gdzie.lewo** i **gdzie.prawo** – musimy napisać graczowi, że jego przygoda się skończyła i że może spróbować jeszcze raz. Aby zacząć grę od początku, jako aktualne pomieszczenie przypisujemy celę – czyli obiekt, z którego rozpoczyna się wędrówkę.

8 Kiedy **gdzie.lewo** i **gdzie.prawo** nie są puste, wypisujemy nazwy pomieszczeń ze znajdujących się tam obiektów, poprzedzając je numerami **A**. Dodatkowo dopisujemy trzecią opcję, czyli **Zawróć**.

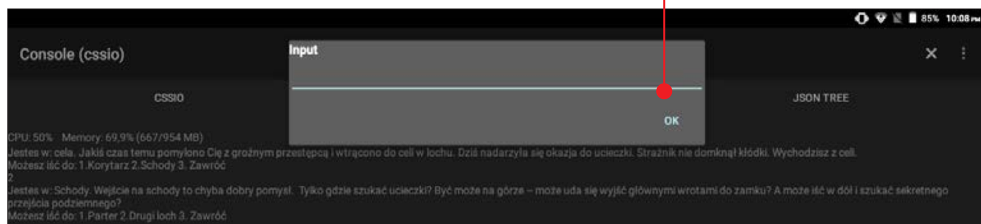
9 Tworzymy też zmienną **wybor** **B** – w typie **Integer**. To do niej zapiszemy numer opcji, którą wybrał gracz.

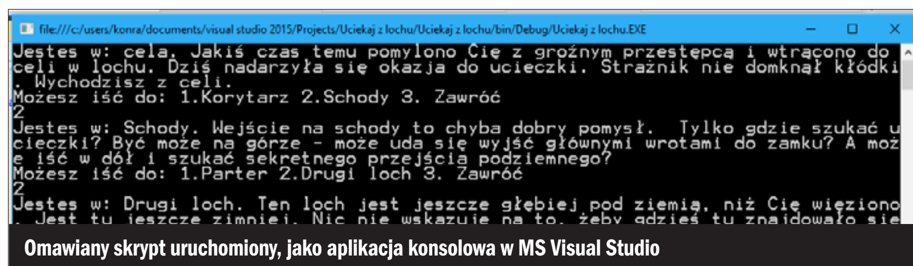
10 Następnie poprzez instrukcję **if** **C** sprawdzamy, jaki był wybór gracza, i jeśli zdecydował się na **1** lub **2** – zapisujemy aktualne pomieszczenie jako poprzednie, a aktualne zmieniamy na wybrane.
Jeśli zdecydował się na **3** (bądź napisał jakąkolwiek inną liczbę) – do aktualnego pomieszczenia przypisujemy pomieszczenie, które było zapisane wcześniej jako poprzednie.

11 Koniec pętli **while** nie oznacza końca funkcji **Main**. Po pętli wypisujemy informację **D** dla gracza, że udało mu się wygrać grę i uciec z lochu.

Uruchamianie i testowanie

Korzystając z programu C# Shell, uruchamiamy skrypt, naciskając przycisk **Play** widoczny w prawym górnym rogu okna programu. Po jego naciśnięciu, jeśli w skrypcie nie pojawiły się błędy, możemy już grać.





Omawiany skrypt uruchomiony, jako aplikacja konsolowa w MS Visual Studio

PROGRAM OPISANY W TYM ROZDZIALE ZNAJDZIESZ NA DVD I W KŚ+

6 C# w Unity – narzędziu do tworzenia gier

C# to popularny język programowania. Wykorzystując go, możemy tworzyć gry i aplikacje za pomocą wielu popularnych narzędzi. Na szczególną uwagę zasługuje fakt, że C# jest szeroko wykorzystywany w wielu silnikach gier

Czym są silniki gier

Gracze nie mają wątpliwości co do tego, że wiele gier jest pod pewnymi względami do siebie podobnych. Podobieństwa te bardzo często dotyczą gier, które oparte są na tym samym silniku. Ale czym właściwie jest silnik gry?
W dużym uproszczeniu jest to narzędzie, które dostarcza nam znaczną część kodu gry, który jest przez ten silnik generowany. O automatycznym generowaniu kodu przez narzędzie mówiliśmy już w przypadku MS Visual Studio. Przecież tworzyliśmy programy oparte na schemacie Windows Forms Application. Wykorzystywaliśmy przyciski czy Panele albo etykiety – a przecież są to elementy, które również trzeba zaprogramować. Jak wiemy z rozdziału trzeciego, dostając do dyspozycji klasę **Form**, otrzymywaliśmy tylko plik z częścią kodu tej klasy. Reszta była generowana przez narzędzie.
Podobnie rzecz ma się z silnikami gier. Dla przykładu: możemy wybrać odpowiedni silnik do gry platformowej, który dostarczy nam na przykład możliwość sterowania postacią i chodzenia nią po platformach, a zadaniem programisty będzie tylko wykorzystanie tej bazy i stworzenie na jej podstawie innych elementów gry.

Unity

Bardzo popularnym silnikiem gier jest **Unity**. Od pewnego czasu jest to narzędzie (z pewnymi ograniczeniami) darmowe. Wersji Personal mogą używać osoby początkujące, uczące się i hobbystycznie tworzące gry. To dobra opcja również dla małych firm – te, których dochód roczny brutto nie przekracza 100 tysięcy dolarów, również mogą korzystać z darmowej wersji silnika gier Unity. Dopiero powyżej tej kwoty są one zobowiązane do przejścia na płatną wersję programu.
Inną ogromną zaletą tego silnika jest to, że pozwala na tworzenie efektownych gier 3D z relatywnie małym nakładem pracy i nie wymaga do tego dużej wiedzy programistycznej. Po zdobyciu programistycznych podstaw podczas realizacji wcześniejszych projektów możemy stworzyć grę w Unity.

Unity – rozpoczynamy pracę z narzędziem

Unity (DVD-KOD: 022) znajdziemy na płycie dołączonej do książki. Po zainstalowaniu programu niezbędne jest zalogowanie się do niego. Możemy zalogować się poprzez nasze konto Google, konto z serwisu Facebook lub poprzez przycisk **create one** przejść do utworzenia specjalnego konta przeznaczonego właśnie do Unity. Logowanie jest niezbędne do uzyskania licencji. Darmowa licencja – to wersja **Personal**. Prócz określenia wersji programu, jaka jest nam potrzebna, zakładając konto, musimy wskazać, czy jesteśmy firmą, która zarabia więcej lub mniej niż 100 tysięcy dolarów rocznie, czy też nie używamy Unity w profesjonalnych celach zarobkowych. Zaznaczamy trzecią opcję.

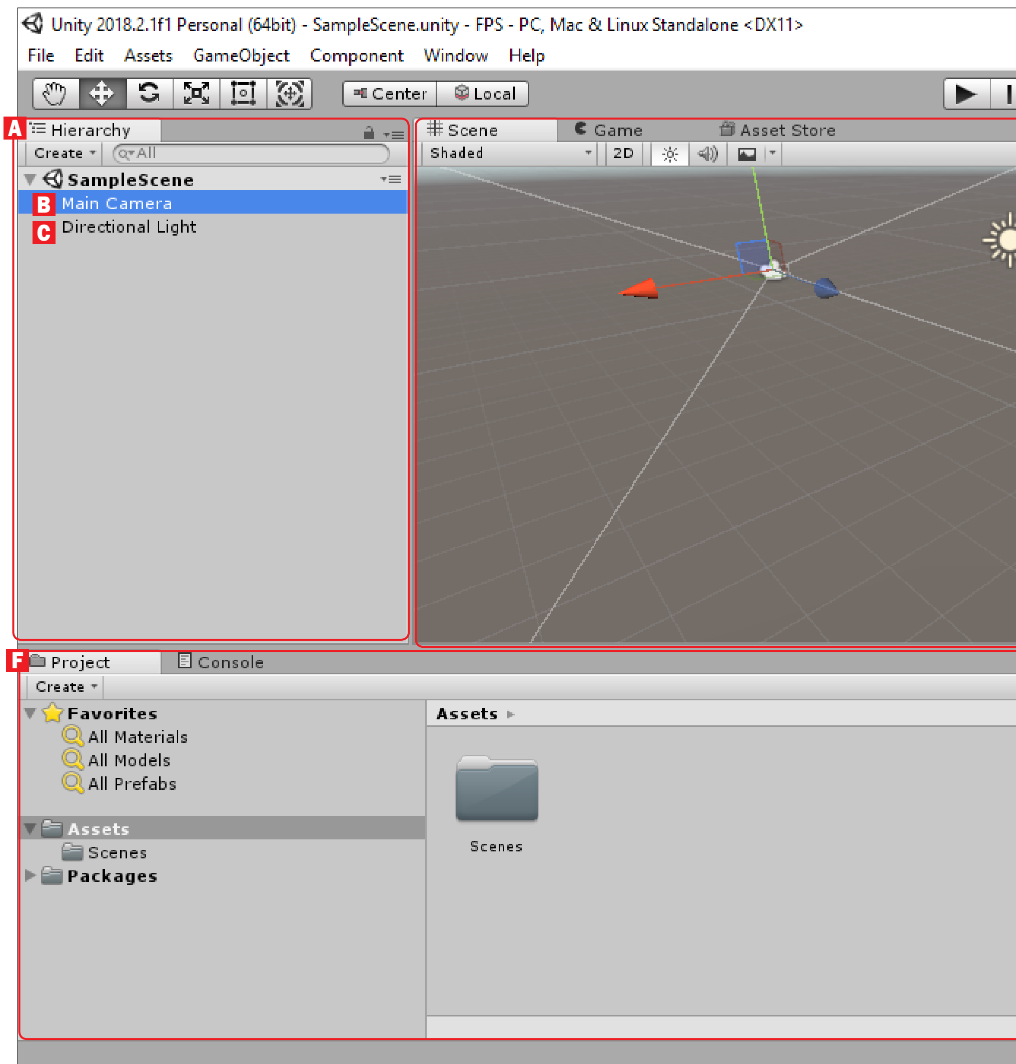
Tworzymy nowy projekt

Kiedy uda nam się już przejść przez proces logowania i uzyskiwania licencji, możemy rozpocząć tworzenie naszego pierwszego projektu. Aby go utworzyć, klikamy na przycisk **New project**. Dalej musimy wskazać nazwę A i lokalizację B projektu.
Unity doskonale sprawdza się, kiedy chcemy tworzyć gry trójwymiarowe z widokiem pierwszoosobowym (tak zwanym First Person). Popularne gry z widokiem pierwszoosobowym to zwykle strzelanki – tak zwane FPS (First Person Shooter). W naszym przykładzie stworzymy grę właśnie tego typu. FPS to gra trójwymiarowa. Stąd ważne jest też odpowiednie ustawienie pola **Template** – wybieramy opcję **3D** C, aby utworzyć projekt oparty na technologii 3D. Kiedy wszystkie ustawienia zostały już wybrane, możemy kliknąć na **Create project** D.

Unity – opis narzędzia

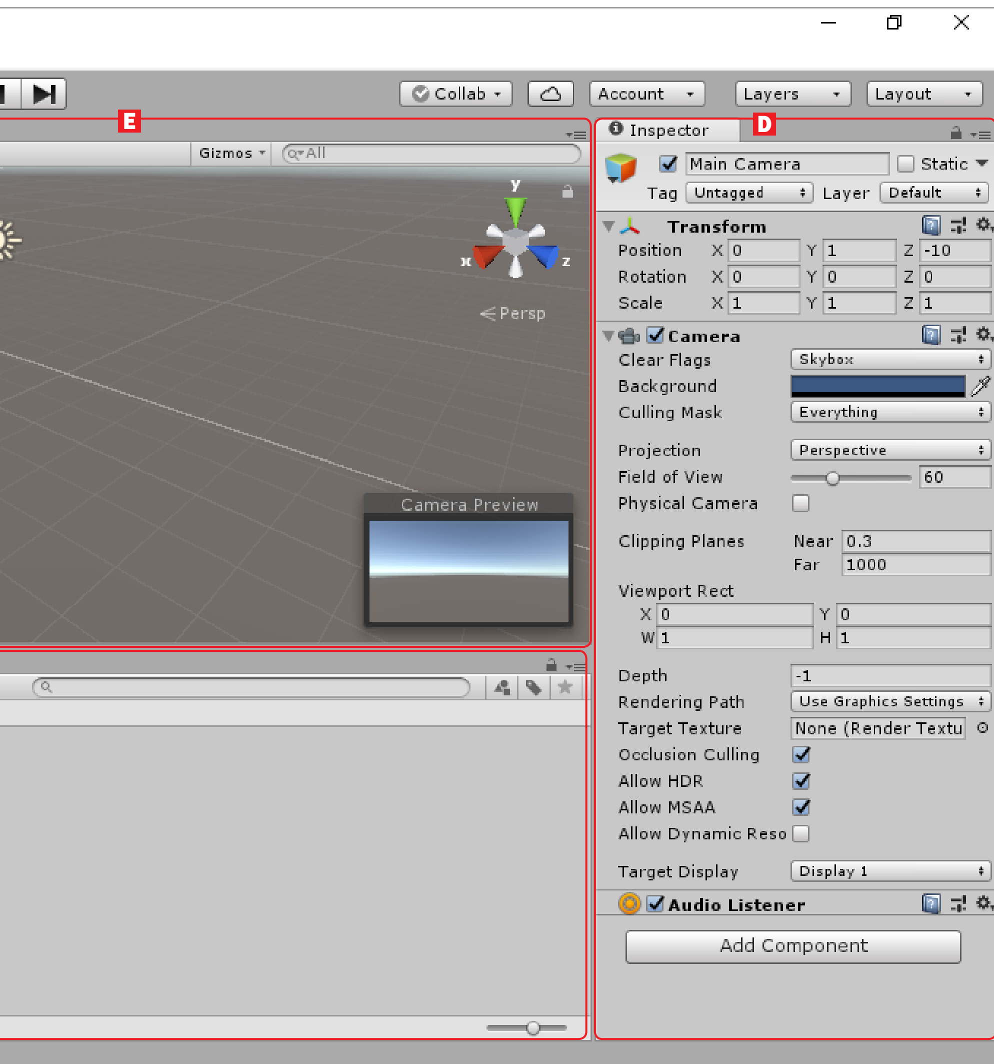
Po utworzeniu projektu mamy już pierwszy plik ze sceną. Zanim jednak przejdziemy do tworzenia gry, warto dowiedzieć się, do czego służą poszczególne sekcje w oknie programu. Po lewej stronie znajduje się sekcja **Hierarchy** **A** – to hierarchia obiektów w grze.

Początkowo na scenie gry są dwa obiekty. **Main Camera** **B** (czyli główna kamera) to obiekt, który pełni rolę „oczu gracza". Tworząc naszą grę i przemieszczając się w niej postacią, będziemy tak naprawdę poruszać kamerą, zmieniając obraz, który jest akurat widoczny



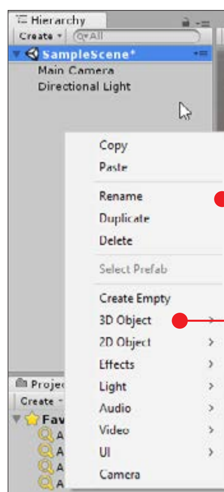
w oknie gry. Drugi obiekt to **Directional Light** **C**, czyli światło, które służy do odpowiedniego oświetlenia obiektów sceny i jest wykorzystywane podczas renderowania gry. Po prawej stronie znajduje się sekcja **Inspector** **D**. Jest ona odpowiednikiem znanej nam z MS Visual Studio sekcji Właściwości. W tej właśnie sekcji wyświetlane są właściwości zaznaczonych obiektów.

Między hierarchią a inspektorem znajduje się **podgląd sceny** **E**. Są na nim widoczne wszystkie dodawane obiekty, to w tej sekcji budujemy świat gry.
U dołu okna programu znajduje się sekcja **Project** **F**, w której znajdziemy menedżer zasobów naszego projektu. Korzystając z niego, możemy umieszczać obiekty na scenie.



Budujemy świat gry

Znając już podstawowy rozkład sekcji w oknie programu, możemy rozpocząć właściwe wykorzystywanie narzędzia. Pierwszym etapem tworzenia naszej gry będzie budowa jej świata, czyli otoczenia, po którym będzie się poruszać nasza postać. Unity wśród obiektów, jakie możemy umieszczać na scenie, oferuje **Terrain** (czyli teren). Wykorzystamy ten obiekt. Dodamy go do naszej sceny.

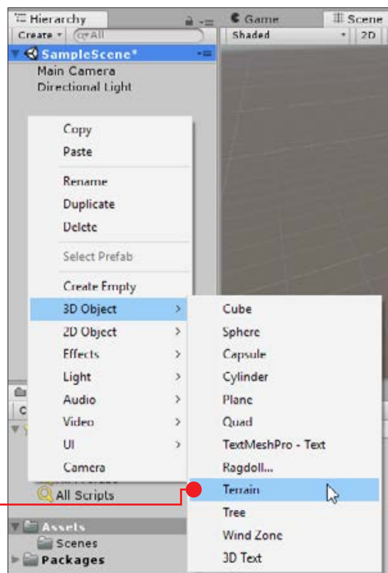


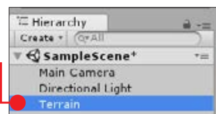
1 Klikamy prawym przyciskiem myszy na pusty teren w sekcji **Hierarchy**. Powoduje to wyświetlenie menu kontekstowego.

2 Z menu wybieramy opcję **3D Object**.

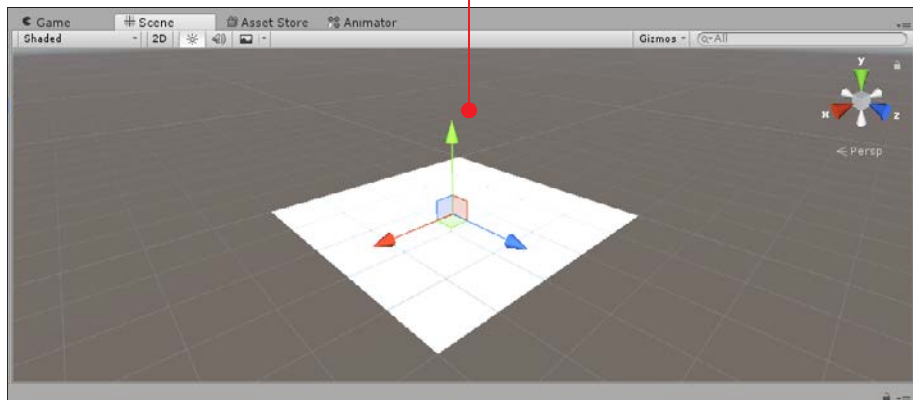
3 Następnie z rozwiniętej listy obiektów wybieramy **Terrain**.

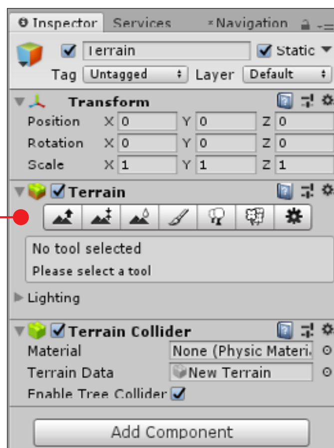
4 Obiekt jest teraz widoczny zarówno w sekcji **Hierarchy**, jak i na podglądzie sceny.





5 Obecnie teren, który dodaliśmy, jest płaszczyzną. Możemy jednak go zmodyfikować. Spójrzmy na właściwości obiektu w sekcji **Inspector**. Znajduje się tam część **Terrain** ze zbiorem



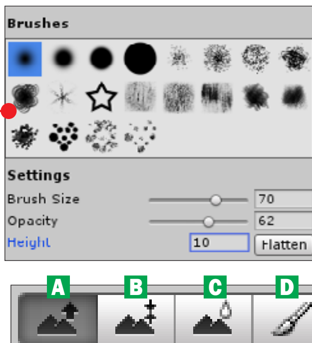


przycisków ● służących do modelowania terenu.

6 Wybranie pierwszego z przycisków daje nam dostęp do zestawu pędzli ●.

7 Należy teraz wybrać jeden z dostępnych pędzli, a następnie, mając wybraną pierwszą opcję z zestawu przycisków **A**, kliknąć na płaski teren w podglądzie sceny. Powoduje to dodanie wybrzuszenia na terenie w miejscu kliknięcia ●.

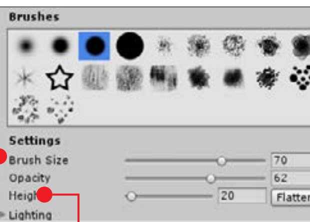
8 Im dłużej będziemy trzymać wciśnięty przycisk myszy, tym wyższe powstanie wzniesienie. Obszar zajmowany przez jedno wybrzuszenie jest definiowany przez rozmiar pędzla. Możemy go zmieniać, przesuwając suwak **Brush Size** ●. A co, jeśli wybrzuszeń już nie chcemy? Jak się ich pozbyć? W takim wypadku również musimy wybrać pierwszy z przycisków, jednak klikając na miejsca, w których chcemy zlikwidować wybrzuszenia, powinniśmy mieć wciśnięty na klawiaturze klawisz shift.

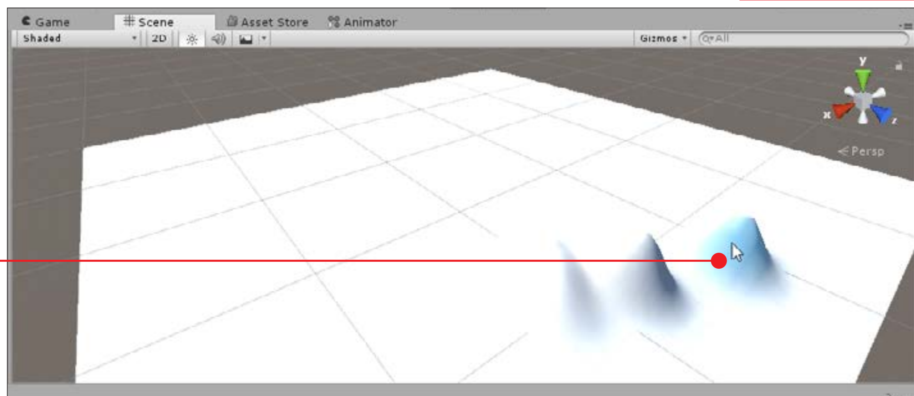


9 Wybrzuszenia powstają jednak tylko, gdy wybraliśmy pierwszy z przycisków w inspektorze. Do czego służą zatem wszystkie przyciski?

A To omawiane już wcześniej tworzenie wybrzuszeń.

B To opcja nieco podobna do tworzenia wybrzuszeń. Polega ona na równaniu terenu do określonej wysokości. Po wybraniu tej opcji we właściwościach pojawia się pole **Height** ●. Wpisana przy nim liczba mówi, do jakiej wysokości chcemy równać teren. Równanie odbywa się tak jak wybrzuszanie – czyli poprzez wciskanie lewego przycisku myszy, gdy kursor znajduje się w miejscu, które chcemy



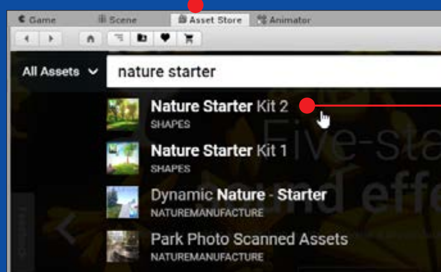


SKĄD WZIĄĆ DRZEWA

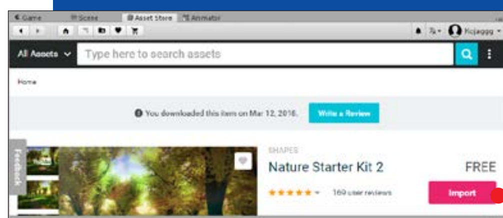
Tu z pomocą przychodzi nam ważny moduł Unity – czyli **Asset Store**. To sklep z zasobami. Z niego możemy pobierać zarówno darmowe, jak i płatne obiekty, aby dodawać je w naszych grach.

1 W Unity sklep jest wyświetlany po wybraniu zakładki **Asset Store** nad podglądem sceny. W pole wyszukiwania powinniśmy wpisać **nature starter kit**, aby odszukać paczkę zasobów zawierającą między innymi drzewa.

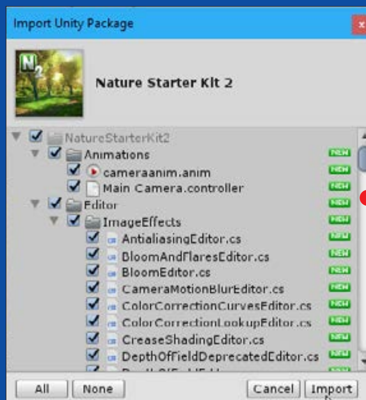
2 Po wyświetleniu wyników wyszukiwania należy wybrać z nich zasób **Nature Starter Kit 2**.



3 Po kliknięciu na niego możemy wybrać opcję importowania – przycisk **Import**.

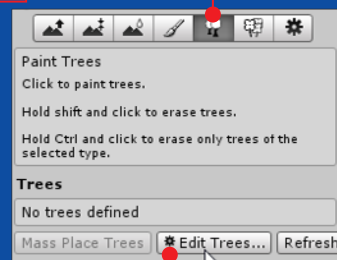


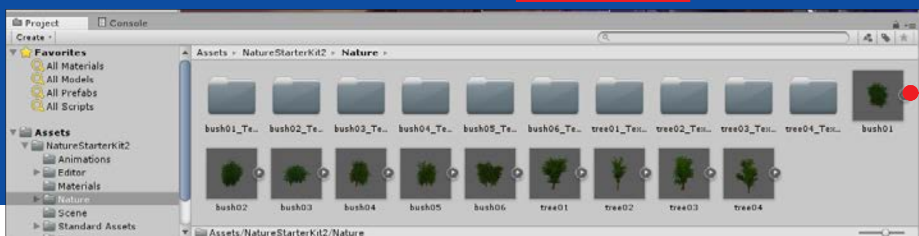
4 Dalej wyświetlona zostanie pełna lista zasobów znajdujących się w paczce. Możemy zdecydować, poprzez zaznaczenie, które z nich chcemy importować. Najlepiej jest jednak pobrać wszystkie.

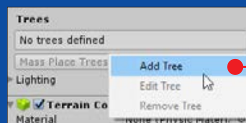


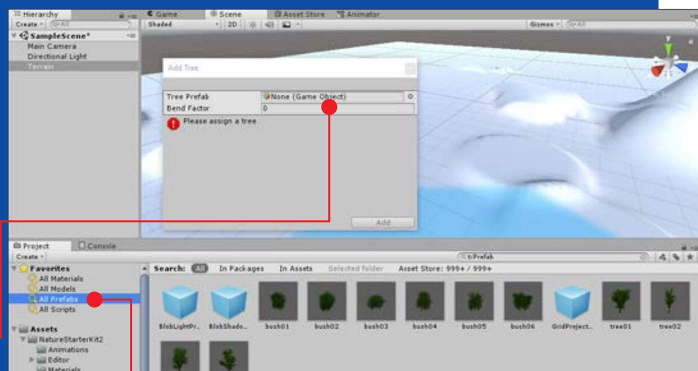
5 Kiedy import się zakończy, nowe zasoby pojawiają się w menedżerze projektu. Są wśród nich drzewa.

6 Następnie należy przenieść drzewa do opcji masowego „zadrzewiania". Po jej wybraniu należy kliknąć na przycisk **Edit Trees**.





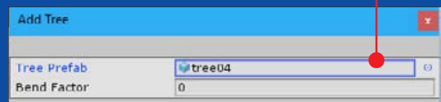




7 Potem z wyświetlonego menu wybieramy opcję **Add Tree** ●.

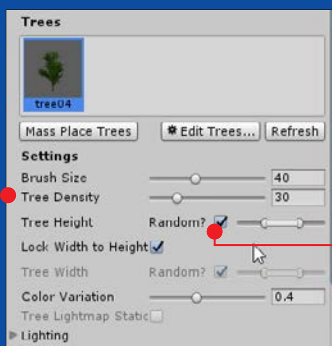
8 Powoduje to wyświetlenie nowego okna. W nim widać pusty slot ● przy pozycji **Tree Prefab**. To właśnie do niego należy przeciągnąć prefabrykat drzewa.

9 Prefabrykaty drzew z pobranej paczki zasobów są teraz widoczne w menedżerze zasobów. Najprościej dostać się do nich poprzez wybranie kategorii **All Prefabs** ●. Po wybraniu interesującego nas drzewa należy przeciągnąć je w pusty slot ●.



10 Kiedy przeciągniemy już drzewo, w oknie inspektora możemy dobrać właściwości masowego zadrzewiania. Wśród nich znajduje się opcja **Tree Density** ●. To gęstość zadrzewiania – podana tam liczba drzew znajdzie się na obszarze oznaczonym przez pędzel (samo zadrzewianie, podobnie jak wybrzuszanie, malujemy po terenie pędzlem o określonym kształcie).

11 Aby drzewa różniły się między sobą, przy opcji **Tree Height** ● należy zaznaczyć znacznik **Random**, co pozwoli na zadrzewianie drzewami o losowych wysokościach.




PRZED UŻYCIEM MASOWEGO ZADRZEWIANIA


PO UŻYCIU MASOWEGO ZADRZEWIANIA

wyrównać. Tworzone w taki sposób wzniesienia są płaskie u góry, gdyż teren nie rośnie ponad określoną wysokość.

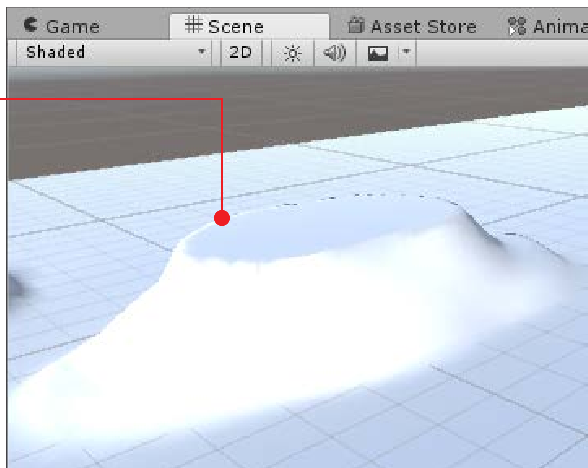
C Trzecia opcja to wygładzanie – jeśli używaliśmy pędzli, które tworzą małe i „ostre" wzniesienia, ta opcja pozwoli nam na zaokrąglenie ich krawędzi.

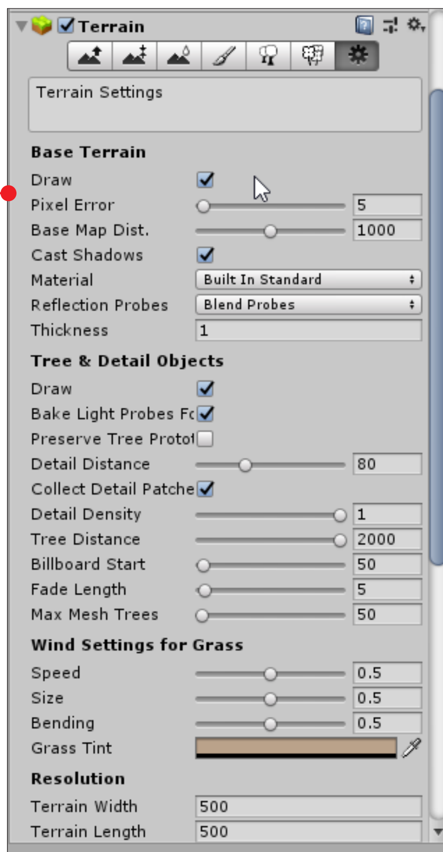
D Czwarta opcja służy do malowania terenu teksturą.

E Piąty z przycisków pozwala na umieszczenie na terenie drzew. To bardzo ciekawa opcja. Wymaga ona jednak, aby w zasobach naszego projektu znajdowały się trójwymiarowe grafiki drzew (patrz ramka na poprzedniej stronie).

F Przedostatni z przycisków to opcja pozwalająca na malowanie detali terenu, takich jak choćby trawy.

G Ostatni z przycisków to ogólne ustawienia terenu – kliknięcie na niego nie daje nam pędzla i możliwości malowania po terenie, ale zestaw opcji do wyboru.





10 Samodzielna praca z narzędziem to doskonały sposób na poznawanie jego obsługi. Dlatego w ramach treningu warto poeksperymentować z modelowaniem i zadrzewianiem terenu – w taki sposób, aby stworzyć świat gry.

KORZYSTANIE Z PODGLĄDU

Podgląd sceny jest modelem trójwymiarowym. To oznacza, że scenę możemy oglądać z różnych perspektyw. Jak zatem „przemieszczać" się po podglądzie sceny?

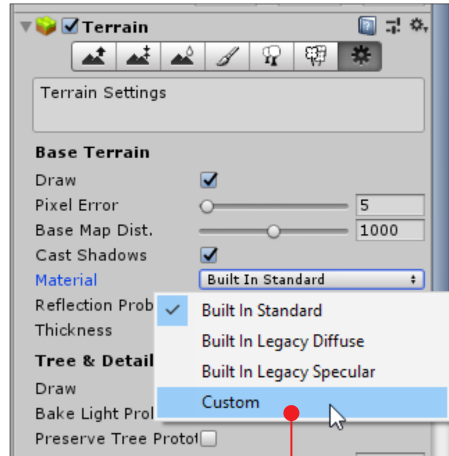
- **Przybliżanie i oddalanie** – kręcenie kółkiem myszy.
- **Przesuwanie** – ruch kursora przy wciśniętym kółku myszy.
- **Obracanie** – ruch kursora przy wciśniętym prawym przycisku myszy.

Malowanie terenu

Po zamodelowaniu terenu możemy przejść do pokrywania go kolorem.

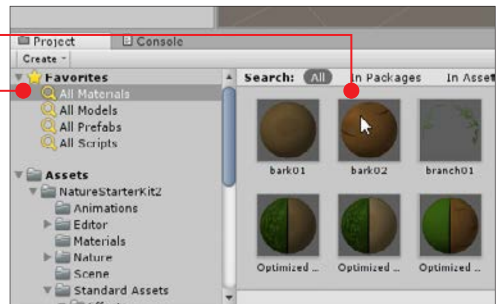
1 Przechodzimy do ustawień obiektu **Terrain**. Przy opcji **Material** zmieniamy domyślne **Built In Standard** na **Custom**.

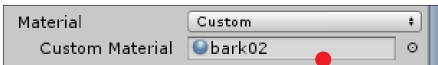
2 Dzięki tej zmianie dostajemy nową opcję **Custom Material** – z pustym slotem.



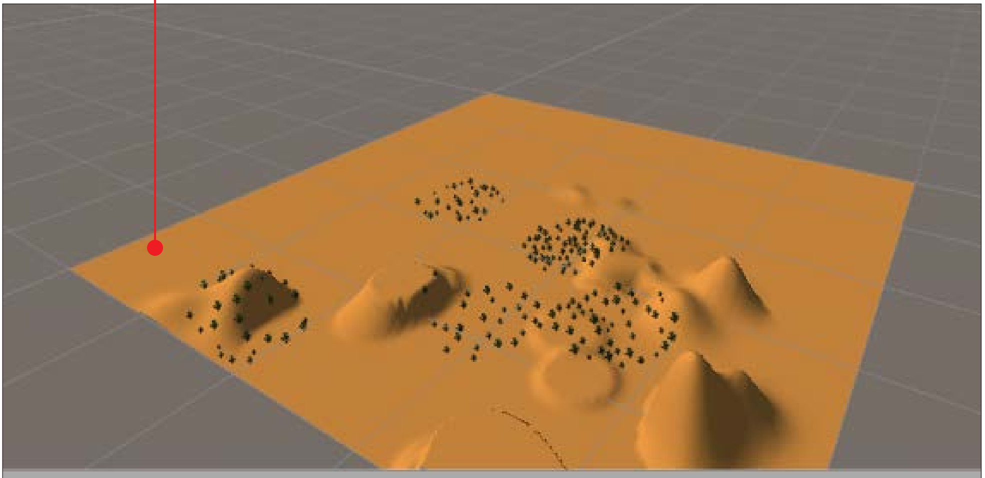


3 Do pustego slotu przeciągamy z menedżera zasobów materiał **bark02** – najprościej go znaleźć, wybierając kategorię **All Materials**.



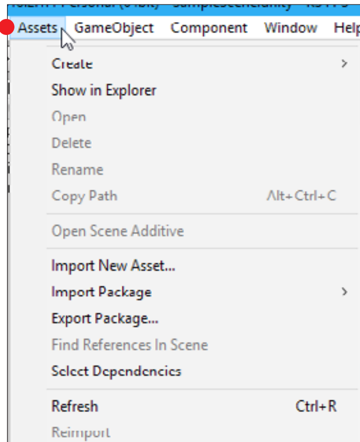


4 Kiedy materiał jest w slocie, na podglądzie sceny widzimy, że teren jest pokryty materiałem, który wygląda jak teksańska ziemia.



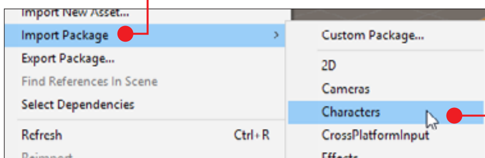
Kontroler postaci

Kiedy teren jest już zamodelowany i pokryty kolorem, możemy postawić na nim postać. Unity ma wbudowany kontroler postaci z widokiem pierwszoosobowym – wykorzystamy go w naszym projekcie. Najpierw jednak trzeba ten kontroler zaimportować do naszego projektu.

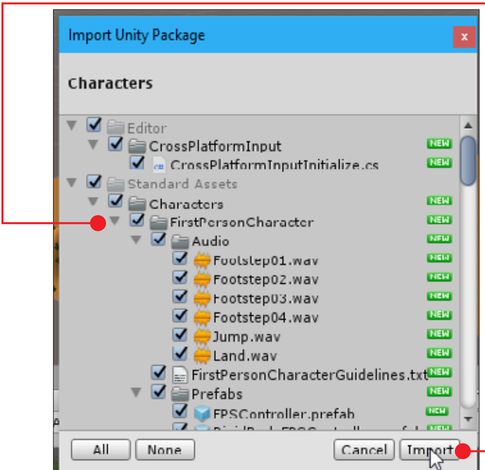


1 Z menu głównego rozwijamy menu **Assets**.

2 W nim odnajdujemy opcję **Import Package**, po której wybraniu możemy zdecydować, jaki rodzaj paczki chcemy importować. Potrzebny nam kontroler postaci znajduje się w paczce **Characters**.

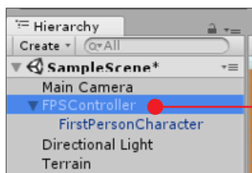


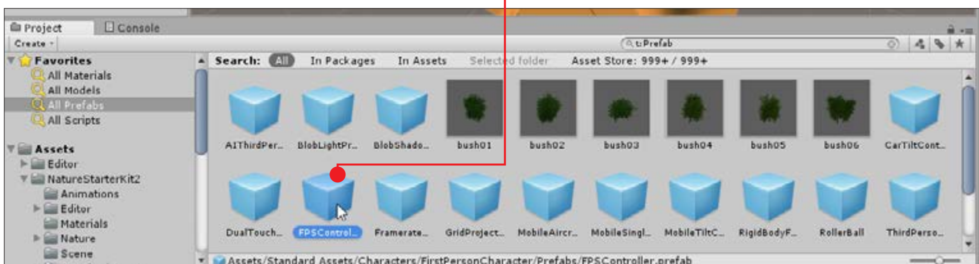
3 Następnie należy wybrać zasoby z paczki, które chcemy zaimportować. Nas interesuje **FirstPersonCharacter**, jednak możemy pozostawić zaznaczone wszystkie zasoby. Importowanie rozpoczynamy, klikając na przycisk **Import**.



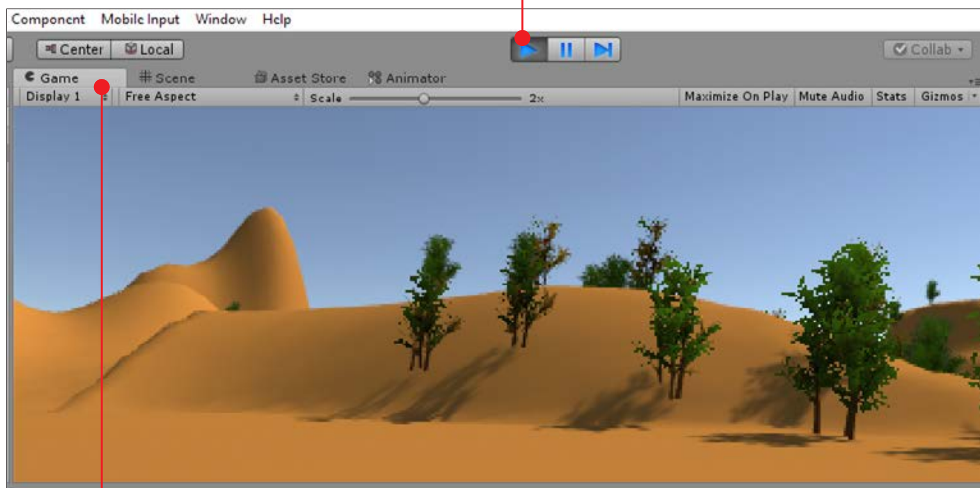
4 Po zaciągnięciu zasobu powinien on pokazać się w menedżerze zasobów projektu. Wybieramy **All Prefabs** i szukamy prefabrykatu **FPSController**.

5 Należy przeciągnąć go na scenę – w miejsce, w którym chcemy, aby stał bohater gry. Po przeciągnięciu prefabrykatu na scenę **FPSController** jest widoczny w hierarchii obiektów.





Pierwsze uruchomienie



Aby zobaczyć, jak wiele daje nam wykorzystanie wbudowanego w silnik kontrolera postaci, możemy po raz pierwszy uruchomić naszą grę. Robimy to, klikając na przycisk **Play** u góry okna programu. Przeniesieni zostajemy wtedy do zakładki **Game**, w której mamy podgląd na naszą grę. Kontroler postaci dostarcza nam między innymi:

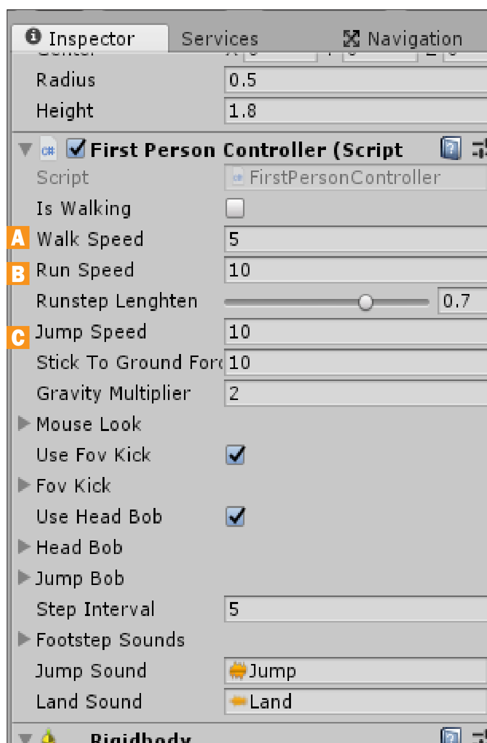
- możliwości chodzenia postacią po terenie
- możliwości obracania postacią poprzez ruch myszy
- możliwości poruszania postacią poprzez klawisze strzałek i WASD
- postać może także biegać (chodzenie + shift) i skakać spacja

W inspektorze obiektu **FPSController** znajdziemy również opcje pozwalające na doprecyzowanie sposobu poruszania się postaci. Znajdziemy tam między innymi:

A **Walk Speed** – prędkość postaci podczas chodzenia,

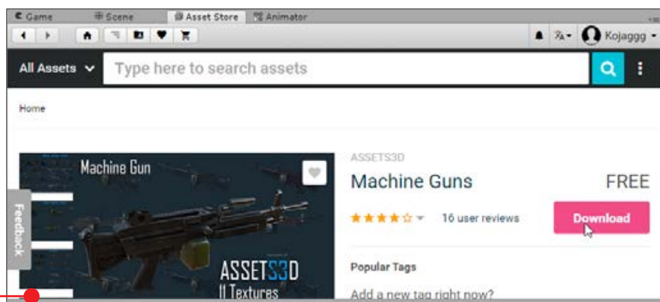
B **Run Speed** – prędkość postaci podczas skakania,

C **Jump Speed** – prędkość skoku, która ma bezpośrednie przełożenie na jego wysokość.



Dodajemy broń do postaci

Tworzona przez nas gra to FPS, czyli First Person Shooter – tak zwana strzelanka. Nie ma mowy o tego typu grze bez broni. Skąd weźmiemy broń dla naszej postaci? Oczywiście z **Asset Store**.

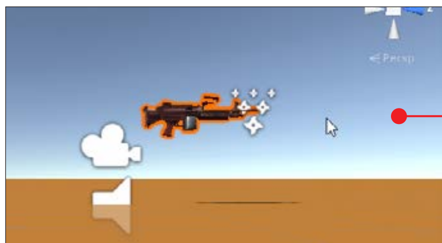


1 Odszukujemy zasób **Machine Gun**, pobieramy i importujemy go do naszego projektu.

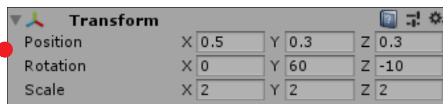
2 Po imporcie w menedżerze zasobów pojawia się folder **FreeMachineGun**, a w nim folder **Prefabs** – w którym znajdują się prefabrykaty broni z pobranej paczki.

3 Wybrany przez nas prefabrykat przeciągamy na **FPSController** w sekcji **Hierarchy**, co powoduje umieszczenie broni na scenie. Musimy jeszcze dobrać odpowiednie położenie i rotację broni, aby wyglądała ona tak, jakby trzymała ją postać gracza.

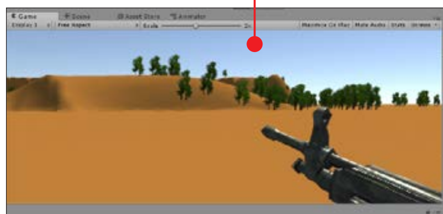


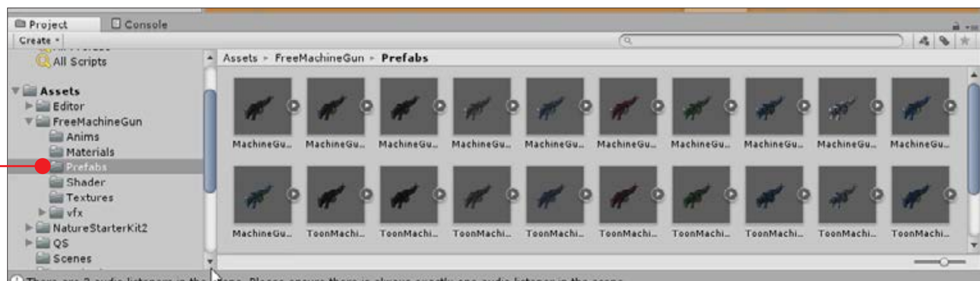


4 W inspektorze obiektu broni zmieniamy jej pozycję, przesuwając ją na osi X o 0,5 punktu, na osi Y o 0,3 punktu i na osi Z o 0,3 punktu. Zmieniamy również jej rotację – na osi Y o 60 stopni, a na osi Z o -10 stopni.



5 Przy kolejnym uruchomieniu podglądu gry nasza postać będzie poruszać się już z bronią.

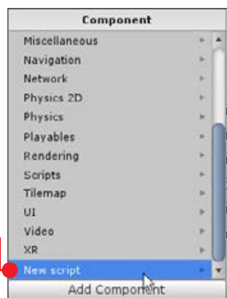




Dodajemy możliwość strzelania

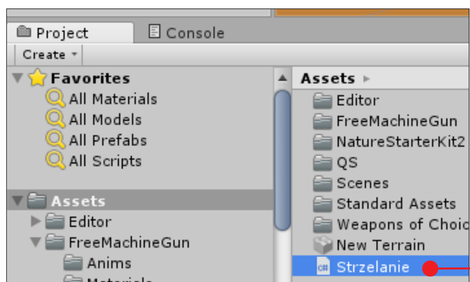
Skoro nasza postać nosi już broń, to aby móc mówić o prawdziwej strzelance, musimy dać jej możliwość strzelania. Aby ją dodać, nie będziemy korzystać w całości z gotowych rozwiązań. Tym razem napiszemy część gry sami. Utworzymy skrypt, który przypiszemy do kontrolera postaci.

1 Zaznaczamy kontroler postaci w hierarchii obiektów i przechodzimy do jego inspektora, gdzie na samym dole znajdziemy przycisk **Add Component**.



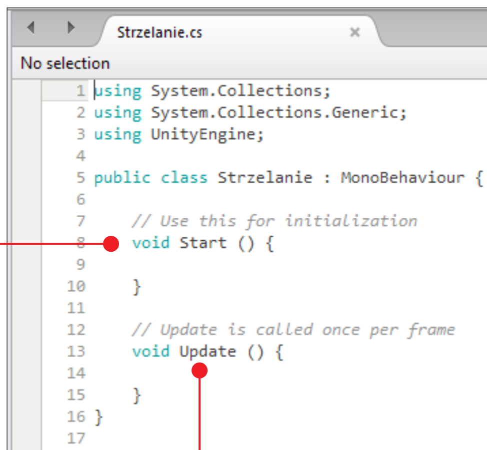
2 Po jego naciśnięciu możemy wybrać kategorię komponentu – w naszym przypadku będzie to **New script**, gdzie po kliknięciu mamy możliwość podania nazwy nowego skryptu. W naszym przypadku będzie to nazwa **Strzelanie**.

3 Po utworzeniu skrypt jest widoczny w menedżerze zasobów. Klikamy na niego dwukrotnie, aby go otworzyć.

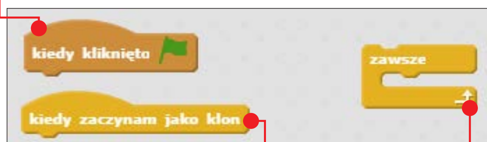


4 Nowy skrypt ma podstawową strukturę skryptu silnika Unity. W utworzonej przez nas w ten sposób klasie **Strzelanie** mamy dwie metody **Start** i **Update**. Aby wyjaśnić ich działanie, po raz kolejny możemy odnieść się do bloków z popularnego Scratcha.

■ Metoda ***Start()*** wykonuje się w momencie pojawienia się obiektu na scenie.



Jeśli obiekt jest na scenie od początku gry, możemy porównać tę metodę z blokiem **Kiedy kliknięto zieloną flagę**. Jeśli obiekt pojawia się na scenie podczas rozgrywki, to możemy metodę tę porównać z blokiem **Kiedy zaczynam jako klon**. To co napiszemy wewnątrz tej metody, zadziałałoby tak, jakby było podpięte do tych bloków. To znaczy wykona się jednorazowo, na samym początku istnienia obiektu.



■ Metoda ***Update()*** doskonale odzwierciedla znaną ze Scratcha pętlę **zawsze**. To oznacza, że metoda ta jest pewnym rodzajem pętli. Instrukcje w niej umieszczone po wykonaniu ostatniej zaraz przechodzą z powrotem do początku i wykonują się ponownie.

5 W naszym skrypcie pozostawimy metodę ***Start()*** nienaruszoną i skupimy się na metodzie ***Update()***. Podsumowując jej działanie: ma ona sprawdzić, czy naciśnięto klawisz odpowiedzialny za strzelanie, następnie sprawdzić, czy i w co udało się trafić, a na końcu umieścić na tym elemencie znak po pocisku. Czy czegoś nam tu nie brakuje? Nie było mowy o wystrzeleniu pocisku i zaprogramowaniu jego lotu. Dlaczego? W tego typu grach wcale nie musi być pocisku! Wystarczy nam złudzenie, że pocisk wyleciał, które powstanie poprzez aplikację efektu trafienia.

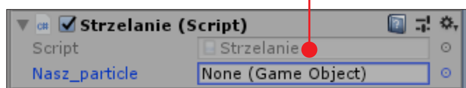
6 Pisanie rozpoczynamy jednak od deklaracji pól, a właściwie jednego pola w klasie. W skrypcie powinno znaleźć się miejsce na pole typu **GameObject**. To typ specyficzny dla silnika Unity. Jest to obiekt gry. W naszym przypadku będziemy przypisywać do tego pola obiekt **particle**. Co to oznacza? W dużym uproszczeniu będzie to grafika, którą umieścimy w miejscu, w które strzelamy. Na razie zadeklarujemy tylko pole w klasie, a dopiero później zajmiemy się dobraniem odpowiedniej grafiki. Jeśli przed deklaracją pola w skrypcie silnika Unity napiszemy **[SerializeField]**, po zakończeniu

```
public class Strzelanie : MonoBehaviour {

    [SerializeField] private GameObject nasz_particle;

    // Use this for initialization
    void Start ()
    {
```

tworzenia skryptu pole to pojawi się w inspektorze obiektu, do którego przypisano skrypt – z wolnym slotem, w którym będziemy mogli (w tym przypadku) umieścić obiekt z grafiką.



7 Przechodzimy teraz do metody ***Update()*** i to właśnie w niej kontynuujemy pisanie. Pierwszą rzeczą, którą musimy zaprogramować, jest sprawdzenie, czy naciśnięto klawisz odpowiedzialny za strzelanie. Dzięki wykorzystaniu możliwości silnika Unity wystarczy, że w metodzie ***Update()*** umieścimy instrukcję warunkową **if**, w której warunku napiszemy ***Input.GetButton(„Fire1”)***,

```
// Update is called once per frame
void Update ()
{
    if (Input.GetButton("Fire1"))
    {

    }
}
```

a sprawdzone zostanie to, czy naciśnięto przycisk, który domyślnie jest klawiszem odpowiedzialnym za strzelanie. Tym domyślnym klawiszem jest lewy przycisk myszy. Tak zbudowana instrukcja warunkowa będzie się wykonywać z każdym kliknięciem lewym przyciskiem myszy.

8 W niej poprzez polecenie ***Ray sprawdzaj = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0f));*** tworzymy obiekty typu **Ray**, który jest niewidzialną wiązką lasera. Następnie wiązkę tę prowadzimy na wprost od środka kamery, tak aby wychwycić, co znajduje się na środku aktualnie wyświetlanego obszaru gry. Deklarujemy również obiekt o nazwie **trafiony** typu **RaycastHit** – posłuży nam on do zapisania w nim obiektu, na który trafiła niewidzialna wiązka.

```
void Update ()
{
    if (Input.GetButton("Fire1"))
    {
        Ray sprawdzaj = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0f));
        RaycastHit trafiony;
    }
}
```

```
if (Input.GetButton("Fire1"))
{
    Ray sprawdzaj = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0f));
    RaycastHit trafiony;

    if (Physics.Raycast(sprawdzaj, out trafiony))
    {

    }
```

```
void Update ()
{
    if (Input.GetButton("Fire1"))
    {
        Ray sprawdzaj = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0f));
        RaycastHit trafiony;

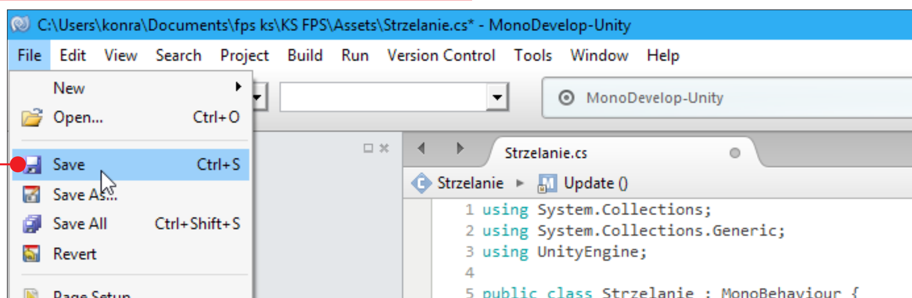
        if (Physics.Raycast(sprawdzaj, out trafiony))
        {
            var efekt = Instantiate(nasz_particle, trafiony.point, Quaternion.identity);
            efekt.transform.rotation = Quaternion.LookRotation(trafiony.normal.normalized);
        }
    }
}
```

9 Wartość do obiektu **trafiony** zostanie przypisana poprzez funkcję **Physics.Raycast(sprawdzaj, out trafiony)**, która z kolei jest funkcją typu boolowskiego – oprócz zwracania obiektu zwraca też informację o tym, czy w ogóle trafiono w obiekt. Możemy ją zatem umieścić w warunku instrukcji **if** i wykonać jakąś czynność, jeśli w ogóle trafimy w obiekt, lub nie wykonywać żadnej czynności, jeśli po kliknięciu lewym przyciskiem myszy nie wykryto żadnego obiektu na środku widocznego obszaru gry.

10 Jeśli zatem wykryto trafienie w obiekt i wiemy już, co to za obiekt, możemy utworzyć nową instancję obiektu graficznego, który przypiszemy później do zadeklarowanego wcześniej pola klasy. Będzie on występował w wielu instancjach – ponieważ każda instancja odpowiada za kolejny strzał i trafienie. Dodatkowo, dla lepszych efektów graficznych, obiekt ten będzie obrócony w taki sposób, aby leżał idealnie na powierzchni, w którą trafił.

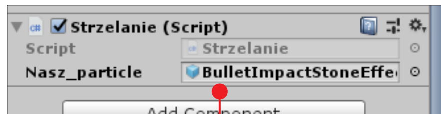
11 Tak napisany skrypt zapisujemy – robimy to, wybierając z menu opcję **File** i **Save**.

12 Możemy zamknąć edytor tekstowy i powrócić do głównego okna Unity. Tam powinniśmy przejść do **Asset Store**

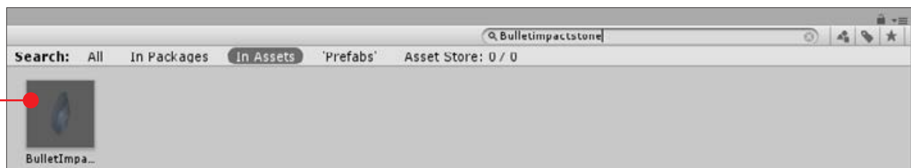


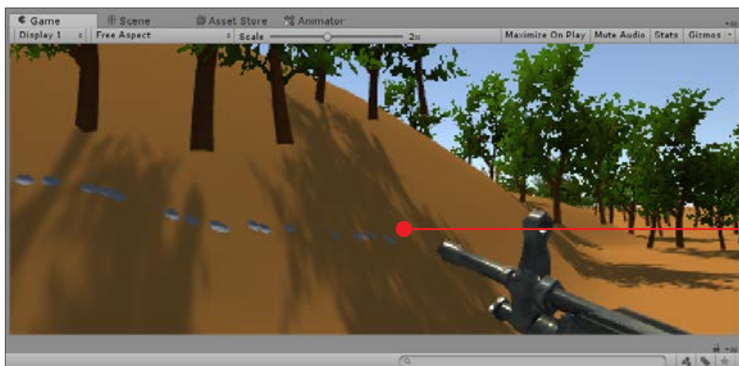


i pobrać paczkę zasobów, w której znajdują się **particle**. Może być to paczka **Unity Particle Pack**.



13 Po jej pobraniu odszukujemy w menedżerze zasobów **BulletImpactStoneEffect** i przypisujemy go do pustego slotu przy skrypcie, w miejscu na obiekt graficzny, który ma być umieszczany jako ślad po pocisku.





14 Następnie możemy uruchomić podgląd gry i oddać kilka strzałów – zobaczymy, że trafienia pozostawiają po sobie ślad!
W ten sposób stworzyliśmy już własny świat i prostą grę typu FPS.

GODOT ENGINE JAKO ALTERNATYWA DLA UNITY

Jednym z zyskujących ostatnio popularność silników gier jest Godot Engine. To rozbudowane narzędzie oferuje nam możliwość tworzenia efektownie wyglądających gier z grafiką 3D. Efekty pracy w postaci gotowej produkcji możemy eksportować między innymi na urządzenia z systemami Windows, Android, OS X czy też z rodziny Linux, a także do HTML5 – aby gra była dostępna w przeglądarce internetowej. Godot przed wersją 3 pozwalał na programowanie jedynie w języku GDScript – stworzonym specjalnie na potrzeby tego silnika.
Ze względu na ogromną popularność języka C# wśród twórców gier, wraz z wersją 3 Godota pojawiła się możliwość programowania również w C#. Ograniczeniem jest jednak brak możliwości eksportu gotowego projektu. Twórcy silnika zapewniają jednak, że jest to etap przejściowy i wraz z kolejnymi wersjami tego oprogramowania taka możliwość się pojawi.

Słowniczek

Framework

Nazywany też **platformą programistyczną** to szkielet do budowy aplikacji. Definiuje on strukturę aplikacji, a także jej ogólny mechanizm działania. Framework dostarcza programiście zestaw komponentów i bibliotek ogólnego przeznaczenia do wykonywania określonych zadań. Programista tworzy aplikację poprzez rozbudowywanie i dostosowywanie poszczególnych komponentów do wymagań realizowanego projektu. W tej książce opisano wykorzystanie .NET Framework.

Klasa

Jest częściową lub całkowitą definicją dla obiektów. Obejmuje dopuszczalne stany i zachowania obiektu. Nie jest samodzielnym bytem – pełni rolę szablonu, który jest wykorzystywany do tworzenia obiektów.

Obiekt

Jest instancją klasy. To podstawowe pojęcie wchodzące w skład paradygmatu obiektowego. Obiekt zbudowany jest z danych i metod, czyli funkcji wykonywanych na danych.

Paradygmat programowania

To pewien wzorzec tworzenia programu, ceniony ponad inne w pewnych okolicznościach lub zastosowaniach. Można powiedzieć, że definiuje on sposób patrzenia programisty na tworzone oprogramowanie.

W programowaniu obiektowym mamy zbiór współpracujących ze sobą obiektów, podczas gdy w programowaniu funkcyjnym definiujemy, co trzeba wykonać, a nie w jaki sposób.

Scratch

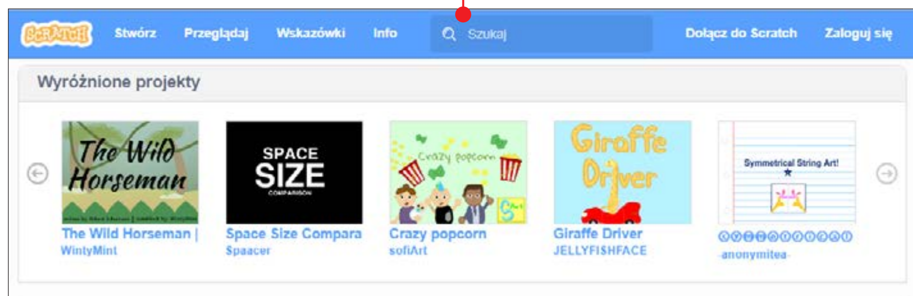
To język programowania, w którym instrukcje przedstawione są w formie graficznych bloków. Jest on szeroko obecny w edukacji programistycznej. Scratch jest również serwisem społecznościowym, w którym każdy zarejestrowany użytkownik może umieszczać stworzone w tym języku programy, dyskutować o nich, a także oglądać i pobierać prace stworzone przez innych użytkowników. O Scratchu przeczytamy więcej w innych książkach z serii Biblioteczka Komputer Świata – 4/2017 i 3/2015.

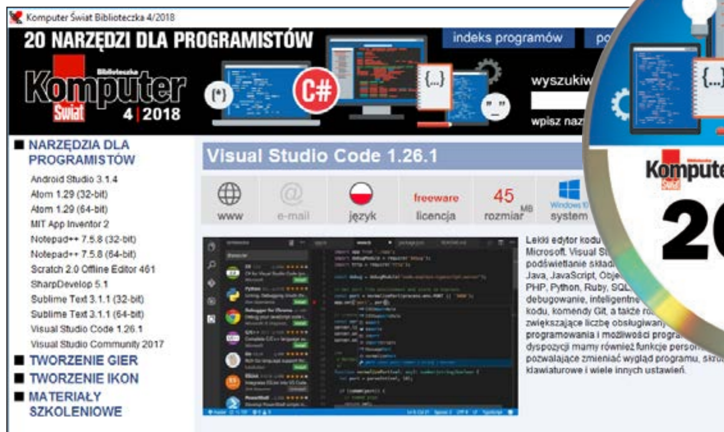


Unity



To silnik gier 2D i 3D zaprezentowany w 2005 roku i tworzony przez Unity Technologies. Można go wykorzystywać, tworząc gry w języku C#.





Programy na płycie

Płyta dołączona do tej książki to prawdziwy niezbędnik programisty. Znajdziemy na niej ponad 20 programów. Oto kilka wybranych

Visual Studio Community DVD-KOD: 025

Podstawowa, bezpłatna wersja środowiska programistycznego Microsoftu do tworzenia nowoczesnych aplikacji na systemy Windows, Android i iOS oraz aplikacji internetowych.

Notepad++ DVD-KOD: 013 / 014 (32- / 64 BIT)

Jeden z najpopularniejszych edytorów tekstu dla programistów i webmasterów. Wyposażony jest w wiele udogodnień związanych z pisaniem i edytowaniem kodu źródłowego, skryptów i stron internetowych. Program podświetla składnię, obsługuje autouzupełnienie kodu, wyszukiwanie i zamianę ciągów znaków za pomocą wyrażeń regularnych, a także wyszukiwanie i podświetlanie par nawiasów (otwierającego i zamykającego).

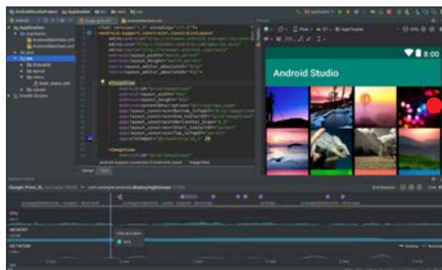
Scratch Edytor Offline DVD-KOD: 016

Scratch to najczęściej wykorzystywane narzędzie edukacyjne, stworzone z myślą o nauczaniu dzieci i młodzieży podstaw programowania. Umożliwia tworzenie i uruchamianie programów napisanych w języku Scratch w trybie offline, czyli bez potrzeby korzystania z internetu. Za jego pomocą można łatwo tworzyć interaktywne historyjki, animacje, gry i muzykę.



Android Studio DVD-KOD: 002

Oficjalne środowisko programistyczne do tworzenia aplikacji mobilnych na system Android opracowane przez Google. Android Studio oferuje narzędzia do edycji kodu, debugowania, testowania oraz profilowania aplikacji. Znajdziemy w nim również emulator pozwalający testować tworzone aplikacje na smartfony, tablety i inne urządzenia z systemem Android.



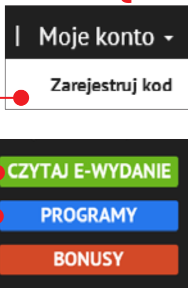
JAK SKORZYSTAĆ Z E-WYDANIA KSIĄŻKI

W KŚ+ znajdziemy **e-wydanie książki, obraz ISO płyty z programami opisanymi we wskazówkach, dodatkowe narzędzia do tworzenia aplikacji na Androida oraz pliki projektów – gier tworzonych w C#.**

Założ konto Logowanie
Zarejestruj kod

1 Otwieramy stronę **www.ksplus.pl**. Logujemy się ● (używamy konta z serwisu **Komputerswiat.pl**). Jeżeli nie mamy konta, klikamy na ●, by się zarejestrować.

2 Po zalogowaniu się możemy zarejestrować kod nadrukowany na płycie dołączonej do książki. Wystarczy kliknąć na link ● i przepisać kod.



3 Uzyskamy w ten sposób dostęp do e-wydania ● i do bonusowego obrazu płyty ● zawierającej narzędzia dla programistów i projekty opisane w książce. Do serwisu KŚ+ możemy logować się zawsze i wszędzie.

UWAGA! W KŚ+ ZA DARMO E-WYDANIE KSIĄŻKI ORAZ PLIK ISO PŁYTY

POLECAMY INNE NASZE KSIĄŻKI, MIĘDZY INNYMI:



MONTAŻ FILMÓW OD A DO Z

Jak nagrywać filmy, montować je i poprawiać darmowymi programami, które znajdziemy na DVD. Dodatki: 200 sampli wideo, muzyka do filmów.



WSZYSTKO O WINDOWS

Pigułka praktycznej wiedzy o Windows: wszystko, co trzeba wiedzieć na wypadek awarii i konieczności reinstalacji, a także o nowej instalacji systemu.



PROGRAMUJ W C#: SAM TWÓRZ GRY 2D I 3D

Nic nie uczy tak jak praktyka. Dlatego w tej książce poznamy instrukcje programistyczne, tworząc według wskazówek krok po kroku gry w C#.

Właśnie na przykładach konkretnych, prostych projektów nauczymy się wykorzystywać programowanie obiektowe, stosować instrukcje warunkowe i pętle, a także zobaczymy, w jaki sposób komunikować się w pisanych przez nas grach z ich użytkownikami – graczami. Najpierw będziemy korzystać z Visual Studio Community 2017, a potem – na deser – przeczytamy, jak ułożyć grę na Androida oraz jak „działa" C# w znanym darmowym narzędziu do tworzenia efektownych gier 2D i 3D – Unity.

Książka ta prezentuje podstawy programowania w języku C# – jednak można dzięki niej zdobyć także uniwersalną wiedzę programistyczną, którą można będzie wykorzystać w przyszłości, poznając kolejne języki programowania.

Na dołączonej płycie znajdziemy najlepsze: edytory kodu, środowiska programistyczne, silniki do tworzenia gier, a także pliki projektów opisywanych w książce.

CENA 14,90 zł
w tym 5% VAT

ISBN 978-83-8091-626-5 INDEKS 321 958



Nr 4/2018 (98)



KOMPUTER ŚWIAT BIBLIOTECZKA